COORDINATED HIGHWAYS ACTION RESPONSE TEAM
STATE HIGHWAY ADMINISTRATION

R1B2 GUI Detailed Design

Contract DBM-9713-NMS
TSR # 9901961
Document #M362-DS-007R0

July 19, 2000
By
Computer Sciences Corporation and PB Farradyne Inc

Revision	Description	Pages Affected	Date
0	Initial Release	All	July 19, 2000

Table of Contents

Table of Figures

1 Introduction

1.1 Purpose

This document describes the detailed design of the Chart II Graphical User Interface (GUI) application for Release 1, Build 2. This design is driven by the Release 1, Build 2 requirements as stated in document M361-RS-002R1, "***CHART II System Requirements Specification Release 1 Build 2***" and further refines the high level design presented in document M362-DS-005, ***"R1B2 High Level Design".***

1.2 Objectives

The main objective of this design is to provide software developers with a framework in which to provide implementation of the software components used to satisfy the requirements of release 1, build 2 of the Chart II system user interface. This document focuses on the client side of each of the system use cases.

1.3 Scope

This design is limited to Release 1, Build 2 of the Chart II system and the requirements as stated in the aforementioned requirements document.

1.4 Design Process

As in the high level design, object-oriented analysis and design techniques were used in creating this design. As such, much of the design is documented using diagrams that conform to the Unified Modeling Language (UML), a de facto standard for diagramming object-oriented designs.

In the high level design, system interfaces were identified and specified. These interfaces were partitioned into logical groupings of packages. This design serves to fill in the details necessary to implement each of the system interfaces identified in the high level design.

In this design, each package identified in the high level design is addressed separately with its own class diagram and sequence diagrams for major operations included in the package's interfaces. Additionally, packages needed for implementation but not present in the high level design are included in this design, with each of these also having its own class diagram and sequence diagrams. Packages are also included for third party software that is needed by the CHART II software, such as the ORB and Java classes. Only classes and methods shown on the sequence diagrams are included in diagrams for third party products.

The design process for each package involved starting with a class diagram including interfaces from the high level design, and filling in details to the class diagram to move toward implementation. Sequence diagrams were then used to show how the functionality is to be carried out. An iterative process was used to enhance the class diagram as sequence diagrams identified missing classes or methods.

1.5 Design Tools

The work products contained within this design are extracted from the COOL:JEX design tool. Within this tool, the design is contained in the Chart II project, R1B2 configuration, SystemDesign phase, A system version is included for each software package.

1.6 Work Products

This design contains the following work products:

- A UML Class diagram for each package showing the low level software objects which will allow the system to implement the interfaces identified in the high level design.
- UML Sequence diagrams for non-trivial operations of each interface identified in the high level design. Additionally, sequence diagrams are included for non-trivial methods in classes created to implement the interfaces. Operations that are considered trivial are operations that do nothing more than return a value or a list of values and where interaction between several classes is not involved.

2 Key Design Concepts

This section provides a high level description of various elements of the design that warrant special attention either due to their technical complexity, central role to system operations, or deviation from previous project practice. For a thorough discussion of how the CHART II GUI fits into the architecture of the CHART II system please refer to the Software Architecture section of document M-361-DS-003R0,
***"CHART II GUI High Level Design For Release 1 Build 1"**.*

2.1 Voice recording, conversion and playback

The CHART II GUI allows an operator to enter a message for activation on a HAR device or storage in a message library in either text or voice format. If the operator opts to record a voice format message, the GUI records the operator's voice via a microphone device attached to the workstation.

The GUI utilizes the javax.sound package to capture the operator's voice. The GUI also allows an operator to listen to the contents of a HAR message using the sound card and speakers attached to his/her workstation. The message is composed of multiple message clips. Each clip is either a text clip, a voice data clip, a voice clip, or a pre-stored message clip. Each of these types of message clips requires slightly different processing in order to be played back to the user in an audible format. A text message clip must be converted into an audible format by the text to speech conversion engine before it can be played. A voice data clip requires no conversion because it contains the binary data required for playback. In this case, the GUI will simply play the data contained in the clip directly back to the user. A voice clip is a message clip used by the system to pass a HAR message around without actually passing the binary audio data. Instead, the clip contains a reference to a CORBA object that can be called to get the binary data via a streaming interface if the data is needed. If the GUI encounters this type of clip, it will call the remote CORBA object and request that the binary data be streamed. A pre-stored message clip is a clip that represents a message that has been stored in a slot on the HAR controller. When the GUI encounters this type of message clip, it will check the configuration object for the HAR and get the message clip stored in the specified slot. The clip in the slot will then be processed as previously mentioned to get the audio data. In all cases, once the GUI has obtained the binary audio data it utilizes the javax.sound package to play the data back for the user through the workstation's sound card and speakers.

2.2 Spell checking

When an operator is entering message text for display on a HAR or DMS device or for storage in a library message, the system will provide assistance in the form of a simple spelling check. The check may be performed at the operator's request while editing the message and will be performed automatically when the operator hits the OK button if the message has been modified since the last spell check. The spell check processing will be performed as follows: Beginning at the start of the text, each word will be checked against a list of approved words in the system. If the word is not a known approved word, a list of (up to three) suggestions will be presented to the user. The suggestions will be composed of the approved words that are lexigraphically closest to the word being checked. The operator will be allowed to ignore the suggestion, ignore all cases where this word exists in the message or replace the word with one of the suggested words. After the user selects his/her option, the system will proceed to the next word and the process will be repeated.

2.3 User and system profiles

In order to allow an operator to modify the working environment of the CHART II system and allow that environment to maintain consistency across workstations, the system will record properties in profiles. The system will use three types of profiles to store preferences; a system profile, a user profile, and a user properties file. A system profile stores properties that pertain to all users regardless of the workstation where they are working. A user profile stores properties that pertain to a particular user regardless of the workstation where the user is logged in. A user properties file stores properties that pertain to a particular user at a particular workstation. In order to make the information in a user or system profile available regardless of the workstation where the user logs in, the properties are stored in the user management database. The user's profile and the system profile are each a collection of key/value pairs. The software functions as follows: When a user logs in his/her user profile and the system profile are retrieved from the user management database and are stored in a temporary Java properties object on the workstation. Property queries are performed against the local properties object for efficiency sake. If the user modifies an existing property or adds a new property, the change is made to the local properties object and to the user or system profile in the user management database as appropriate. Thus, the modifications are available in the profile the next time the user logs in at any workstation.

2.4 Factory choice when creating new objects

When an operator is adding a new object to a distributed system, it is necessary to determine which service should be responsible for serving the new object. In the CHART II system, each service that is capable of serving a particular type of object publishes a factory in the CORBA trading service. Each factory is named according to the district or region from which it is served. For maximum configuration flexibility, the GUI will present the user with this list of factories on the object creation dialog. If the operator selects a particular factory, the system attempts to create the new object using that factory. Any errors encountered are reported to the user and no further processing is performed until the operator takes further action. Along with the list of factories, the operator is presented with an option to allow the software to 'Auto-select' a factory to add the new object to. If this option is selected, the software attempts to add the new object to each factory in the system, in succession, until it is successfully added to a factory. An error is reported if and only if no factory in the system can create the new object.

2.5 Error Processing

Because CHART II is a distributed object system, it is expected that any call to a remote object could cause a CORBA exception to be thrown. All software calls to remote objects handle CORBA exceptions and the processing is not shown on sequence diagrams within this design except where it serves to illustrate a design point.

Additionally, CHART II object interfaces explicitly declare exceptions that may be thrown when a particular method is invoked. All CHART II defined exceptions contain information that can be displayed to the user as well as debugging information. The CHART II GUI handles errors in the following manner. All user displayable error information is displayed to the user in a status pane at the bottom of the active dialog box or in the command status window if no dialog is available. The GUI also utilizes the Log utility class to maintain a flat file that contains debugging information. Each entry in the file contains the name of the class that logged the entry, the date and time the entry was logged, and descriptive text of the error that occurred. The log utility also provides the ability for a stack trace to be printed to the file to accompany the error. This feature is reserved for use when an error condition is caught and the exact cause of the error condition is not known, or when it is known that the caller of the method performing the log passed invalid data. Log files created by the Log utility class are self-cleaning and are automatically removed from the system when they reach a certain age, as specified in a configuration file.

The CHART II GUI also adds a software communications failed state to each remote object. This state is used to indicate that a remote object was not reachable the last time that the system attempted to communicate with it. This information is essential in a distributed software system where objects become unavailable temporarily due to server or network outages. The GUI tracks this state as follows. Each time the GUI attempts an operation on a remote object, if the object cannot be reached, it will be put into a software communications failed state. If the object can be reached, regardless of whether the operation succeeds or throws a defined exception, the GUI marks the object as not being in a software communications failed state.

2.6 Installable Modules

The CHART II GUI application has been designed as a core GUI module and a collection of installable modules. The core GUI module provides access to the services and data needed by any installable module and also provides the main windows of the application. Each installable module adds a coherent set of functionality and windows that are not required by other modules. This design serves to break the application into understandable packages and has the added benefit of allowing for scaled down deployments with limited functionality. At initialization the core GUI module reads a Java properties file and determines which installable modules should be instantiated. The core GUI module then coordinates the activities of each of the installed modules while the application is running. Significant events, such as a user logging in or out of the application, are passed along to each installed module providing them an opportunity to perform cleanup activities.

Throughout the design, there are instances where a particular module needs to access the services of another installable module. This type of coordination typically involves registering one module as a supporter of another module. A good example of this can be observed in the plan module. The plan module can create a plan without the support of any other module. However, it cannot add any items to the plan because the DMS or HAR installable modules must create them. Thus, the plan module provides an API for other modules to call to register as a plan item creation supporter. The plan module may then delegate creation of new plan items to the installed creation supporters. In all cases where a module relies on a call to another installable module the call is made with the expectation that an exception may be thrown indicating that the other module does not exist. If this is the case, the calling module will handle the exception and continue processing as normal. The services provided by the called module will not be available during this instance of the GUI application because the module was not installed.

2.7 Startup and Shutdown

In order to startup correctly, the CHART II GUI requires a CORBA trading service. It will search the trading service for the OperationsCenter object that it will be utilizing to allow a user to login. Both the location of the CORBA trading service and the name of the operations center that the user will log in at are configurable in the GUI properties file which resides in the GUI directory after installation. If the trading service is not available, or the desired operations center is not available, the GUI will issue an error message to the operator and will allow the user to shut it down. The GUI has no other dependencies on external services, nor does it have any dependencies on the order in which installable modules are installed. At startup the GUI will read the properties file and will construct an instance of each of the installable modules listed. After all modules have been constructed, the startup method of each module is called. This guarantees that when any particular module’s startup method is being performed, any other modules that it interacts with will have already been constructed. At shutdown, the GUI calls the shutdown method of each installed module. After shutting down all modules, it deactivates the ORB and POA and exits.

2.8 Packaging

This software design is broken into many packages of related classes. The table below shows each of the packages along with a description of each.

CORBAUtilities	This package contains classes included in the third party ORB product used for implementation. Only classes that are directly referenced from diagrams for CHART II software are included in this package's diagrams.
DataModel	This package includes classes that are used to provide an implementation of the subject observer design pattern.
DataTransfer	This package contains classes that are used to support drag and drop operations in the CHART II GUI.
DMSUtility	This package contains utility classes that are shared among the server and GUI DMS modules. Examples of DMSUtility classes are the MultiConverter and implementation of value types defined in the DMSControl system interfaces.
GUI	This package contains classes that are core to the CHART II GUI application such as the main GUI toolbar window, the command status window and the command failures window.
GUIDMSModule	This package contains an installable GUI module that provides all DMS related functionality.
GUIDictionaryModule	This package contains an installable GUI module that provides all dictionary-related functionality.
GUIHARModule	This package contains an installable GUI module that provides all HAR and SHAZAM related functionality.
GUIMessageLibraryModule	This package contains an installable GUI module that provides all message library related functionality.
GUIPlanModule	This package contains an installable GUI module that provides all plan-related functionality.
GUIResourcesModule	This package contains an installable GUI module that provides all shared resource related functionality. This includes the transfer shared resources dialog.
GUITrafficEventModule	This package contains an installable GUI module that provides all traffic event related functionality. This package also includes the client side functionality for the Comm. Log.

GUIUserManagementModule	This package contains an installable GUI module that provides all user management related functionality.
GUIUtility	This package contains classes that are used by many installable GUI modules.
HARUtility	This package contains HAR related utility classes shared by the server and GUI.
JavaClasses	This package contains classes included in the Java programming language. Only classes that are directly referenced from diagrams for CHART II software are included in this package's diagrams.
Navigator	This package contains classes that implement the Navigator window of the CHART II GUI.
SHAZAMUtility	This package contains SHAZAM related utility classes shared by the server and GUI.
SystemInterfaces	This package contains the CORBA interfaces and related definitions for the CHART II system. These interfaces and classes define the IDL for the CHART II system.
Utility	This package contains utility classes shared by other packages, including classes used to access the database and the OperationsLog class.

The remainder of this document contains detailed designs of each of the above packages.

3 Package Designs

3.1 CORBAUtilities

3.1.1 Class Diagrams

3.1.1.1 CORBAClasses (Class Diagram)

The CORBAUtilities package exists to provide reference to classes that are supplied by the ORB Vendor and are referenced by other packages' class or sequence diagrams.

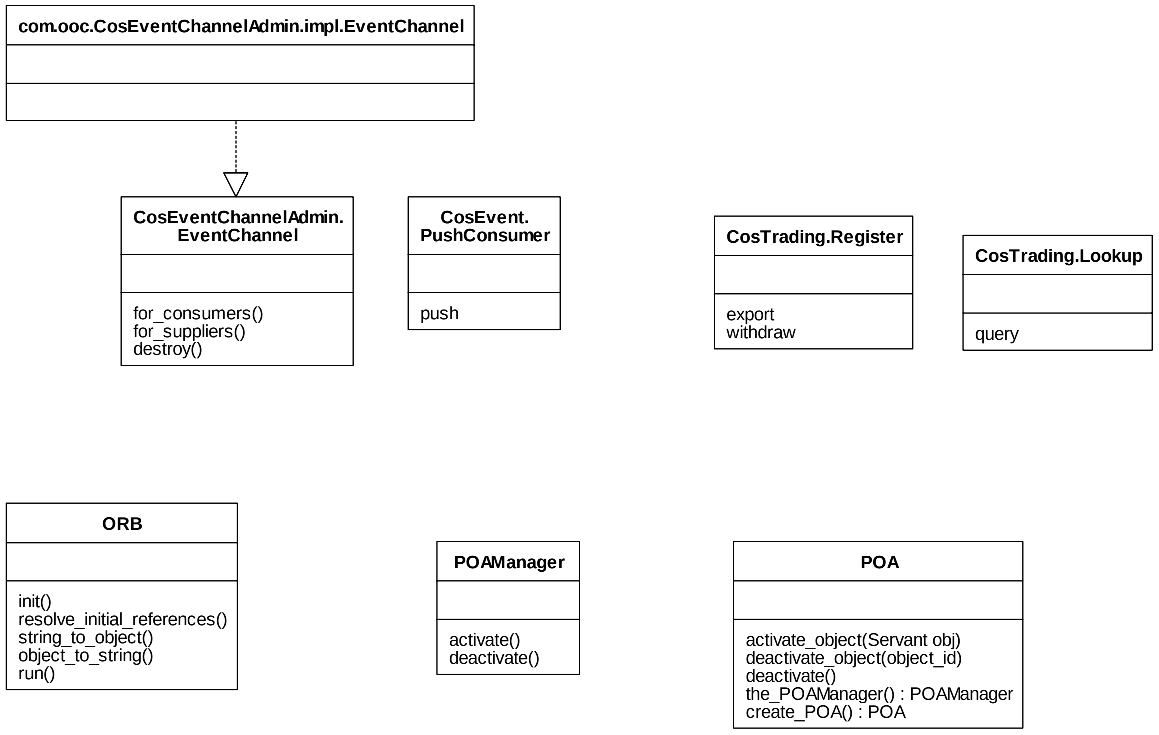


Figure 1. CORBAClasses (Class Diagram)

3.1.1.1.1 com.ooc.CosEventChannelAdmin.impl.EventChannel (Class)

This class is the ORB vendor's implementation of a CORBA event channel. The event service provided by the vendor simply serves one of these objects. The Extended Event Service serves a factory that allows multiple instances of the vendor supplied event channel to be created.

3.1.1.1.2 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.1.1.1.3 CosEventChannelAdmin. EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

3.1.1.1.4 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

3.1.1.1.5 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

3.1.1.1.6 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

3.1.1.1.7 POA (Class)

This interface represents the portable object adapter used to activate and deactivate servant objects.

3.1.1.1.8 POAManager (Class)

This interface represents the portable object adapter manager used to activate and deactivate the POA.

3.2 DataModel

3.2.1 Class Diagrams

3.2.1.1 DataModelClasses (Class Diagram)

The data model classes represent a collection of objects which, when altered via the DataModel, will notify observers that they have been modified. The notification will be delivered in the form of a call to the observer's update() method and will include a collection of changes that have occurred in the system in the preceding interval. Each change is either an object added change, an object removed change, or an object updated change. If the change is an object updated change it may include hints that help an observer determine if it needs to take any action based on the change.

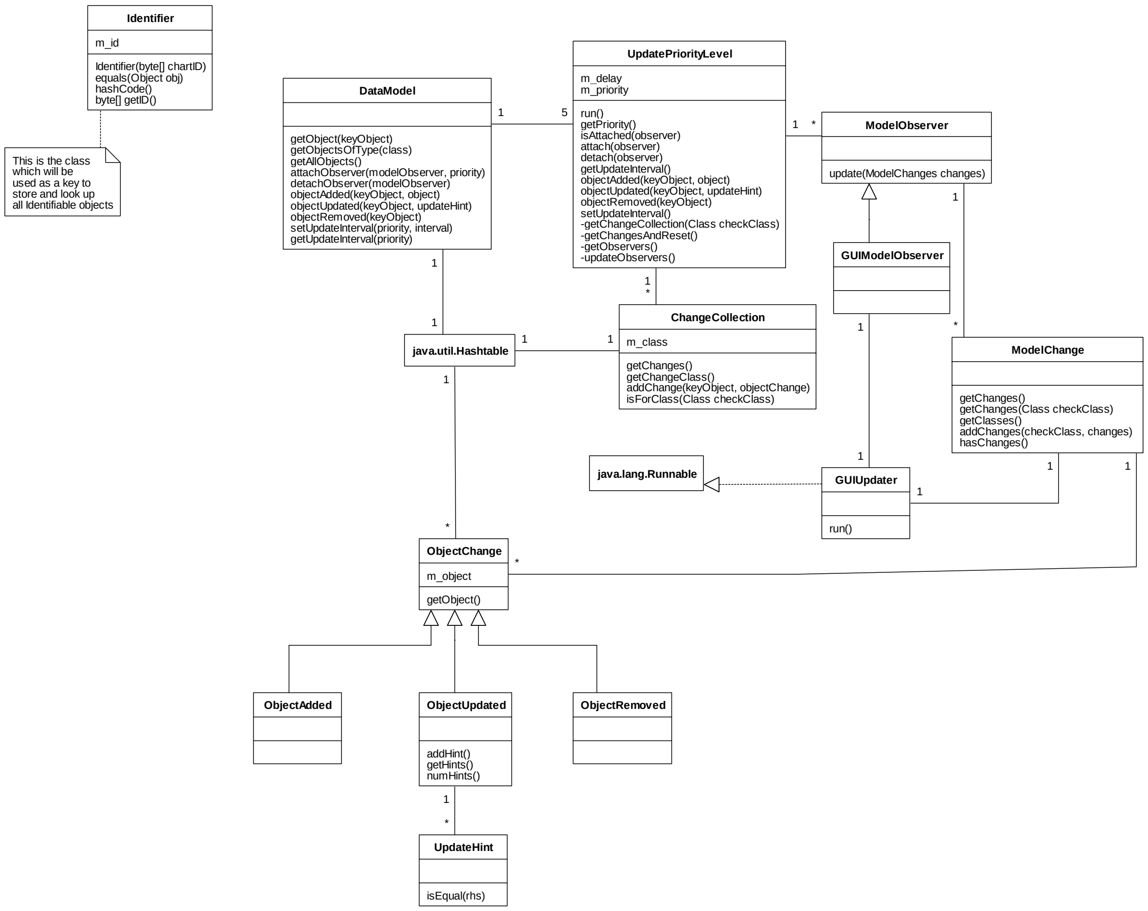


Figure 2. DataModelClasses (Class Diagram)

3.2.1.1.1 ChangeCollection (Class)

This class represents a collection of object changes. All object changes in the collection must be for objects of the same type. Object type is determined by making the Java call getClass(). This allows an observer to look at one object in the collection and determine if it is interested in changes to this type of object. If the observer is not, it may ignore the entire collection.

3.2.1.1.2 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.2.1.1.3 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

3.2.1.1.4 GUIUpdater (Class)

This class is used to send all changes to GUIModelObservers in the GUI event dispatch thread. It does this by storing the changes until the dispatch thread calls the run() method.

3.2.1.1.5 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

3.2.1.1.6 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java’s threading mechanism.

3.2.1.1.7 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

3.2.1.1.8 ObjectRemoved (Class)

This class is used to indicate that the object it represents was removed from the DataModel.

3.2.1.1.9 UpdateHint (Class)

This interface must be implemented by all objects that are to be used as update hints. An update hint is a concept that is negotiated between a (subject) object and observers that are interested in that object. The data model makes no assumptions about how the hints will be used. The data model will invoke the isEqual method of the update hint to ask it to determine if it is equivalent to another hint. This allows the model to perform update optimizations by not sending notification to observers of two updates with equivalent hints in the same period. An example of how an update hint would be used follows: A DMS object has state variables that track the current message being displayed and the current latitude and longitude location of the sign controller. Because the system map requires significant processing load to redraw and needs only be notified if the latitude or longitude of the DMS changes the DMS and map view use a DMSMapChange hint. When the DMS object has a state change to the latitude or longitude property to report, that change is reported by calling objectUpdated and passing a DMSMapChange hint. When it has other changes that are not state changes to the latitude or longitude properties, it reports those changes to the DataModel by calling objectUpdated passing a DMSNonMapChange update hint. The map view will only redraw the DMS if the ObjectUpdate contains a DMSMapChange hint.

3.2.1.1.10 ModelChange (Class)

This class is used to convey changes to observers of the DataModel. It contains all ObjectChanges for a particular update priority level for a particular period of time.

3.2.1.1.11 ModelObserver (Class)

This interface must be implemented by any object that would need to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

3.2.1.1.12 ObjectAdded (Class)

This class is used to indicate that the object it represents was added to the DataModel.

3.2.1.1.13 ObjectChange (Class)

This class represents the changes to a particular object stored in the DataModel for a particular period. The change may be that this object was added to the model, removed from the model, or updated during this period.

3.2.1.1.14 ObjectUpdated (Class)

This class indicates that an object that was already in the model has been updated. The update may be specific to certain parts of the object, and the UpdateHint objects are used to specify which data members within the object were changed. If there are no hints in the ObjectUpdated, it signifies that the entire object has been changed so the observer must query the object for any data members that it is displaying.

3.2.1.1.15 UpdatePriorityLevel (Class)

This class represents a particular priority update level. When an observer attaches to the data model an update priority level is specified. The system currently supports five levels of priority ranging from real time updates for animated displays to delayed updates for windows which can tolerate not being notified for a significant period of time when a change occurs to the system data model. Each time an object is modified it is added to the ChangeCollection for all priority levels. The notification of observers simply happens at longer and longer intervals as the priority level decreases. Thus, an observer of the data model connected at real time may be updated three times in one second while a lower priority observer may only be updated once at the end of the second. However, both observers will be told about the exact same changes that occurred during the second.

3.2.2 Sequence Diagrams

3.2.2.1 DataModel:AttachObserver (Sequence Diagram)

This diagram shows how an observer is attached to the DataModel for the purpose of receiving updates. The DataModel's attachObserver method is called, and if the priority level is supported by the DataModel, the observer will be attached at that priority level. The result of this is that the observer will be updated periodically (with the period depending on the priority level) after changes are made to the objects through the DataModel.

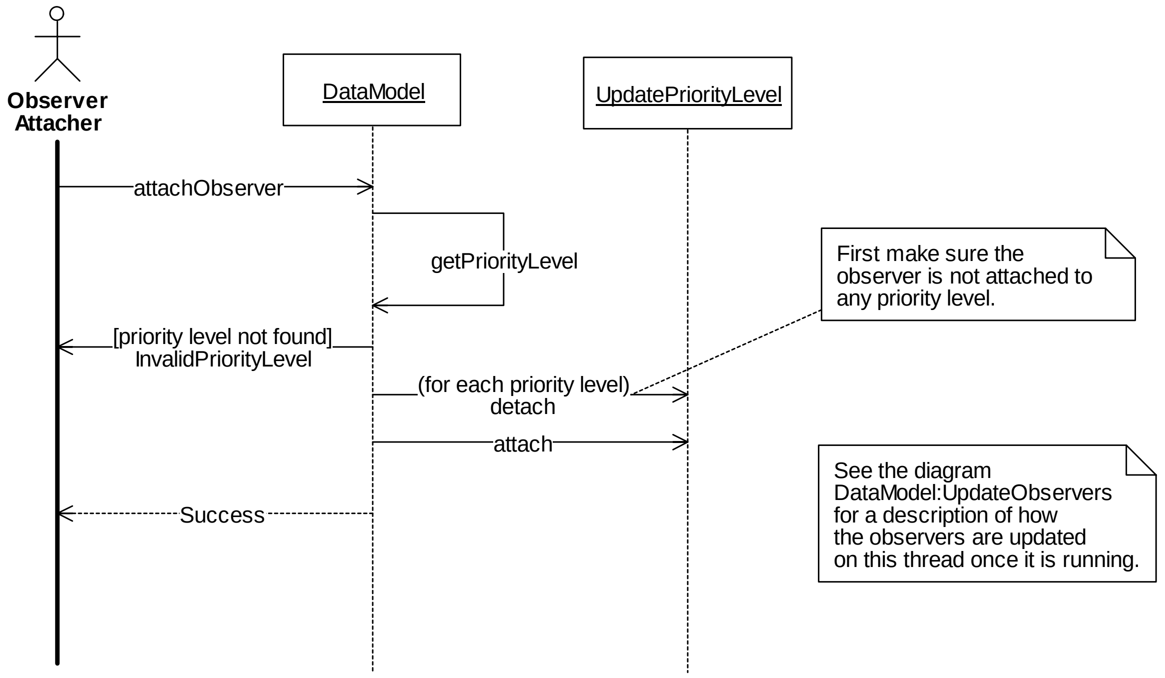

Observer Attacher
DataModel
UpdatePriorityLevel
attachObserver
getPriorityLevel
First make sure the observer is not attached to any priority level.
[priority level not found] InvalidPriorityLevel
(for each priority level) detach
attach
See the diagram DataModel:UpdateObservers for a description of how the observers are updated on this thread once it is running.
Success

Figure 3. DataModel:AttachObserver (Sequence Diagram)

3.2.2.2 DataModel:ObjectAdded_ (Sequence Diagram)

This diagram shows the steps taken when an object is added to the DataModel. First, the Object and the Key are passed into the DataModel's objectAdded method. The DataModel checks whether the object was added before, and if so, the object will not be added again. The DataModel then calls each of the PriorityLevel objects' objectAdded methods so that observers of all priority levels can be updated independently. The PriorityLevel object then checks its ChangeCollection objects to see if a ChangeCollection exists for the class of object which is being added. If not, it will create a ChangeCollection to store all changes for that class. The PriorityLevel then creates an ObjectAdded object to represent the change, then adds it to the ChangeCollection.

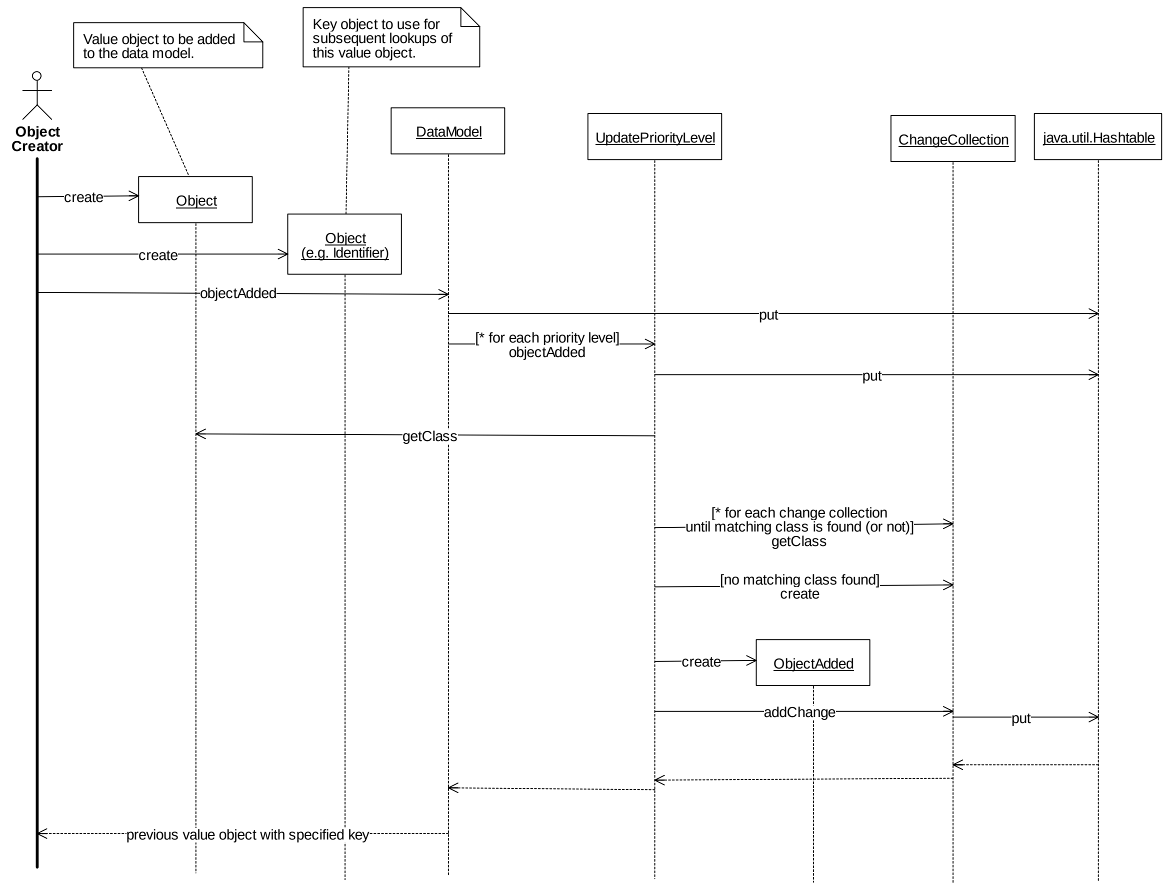
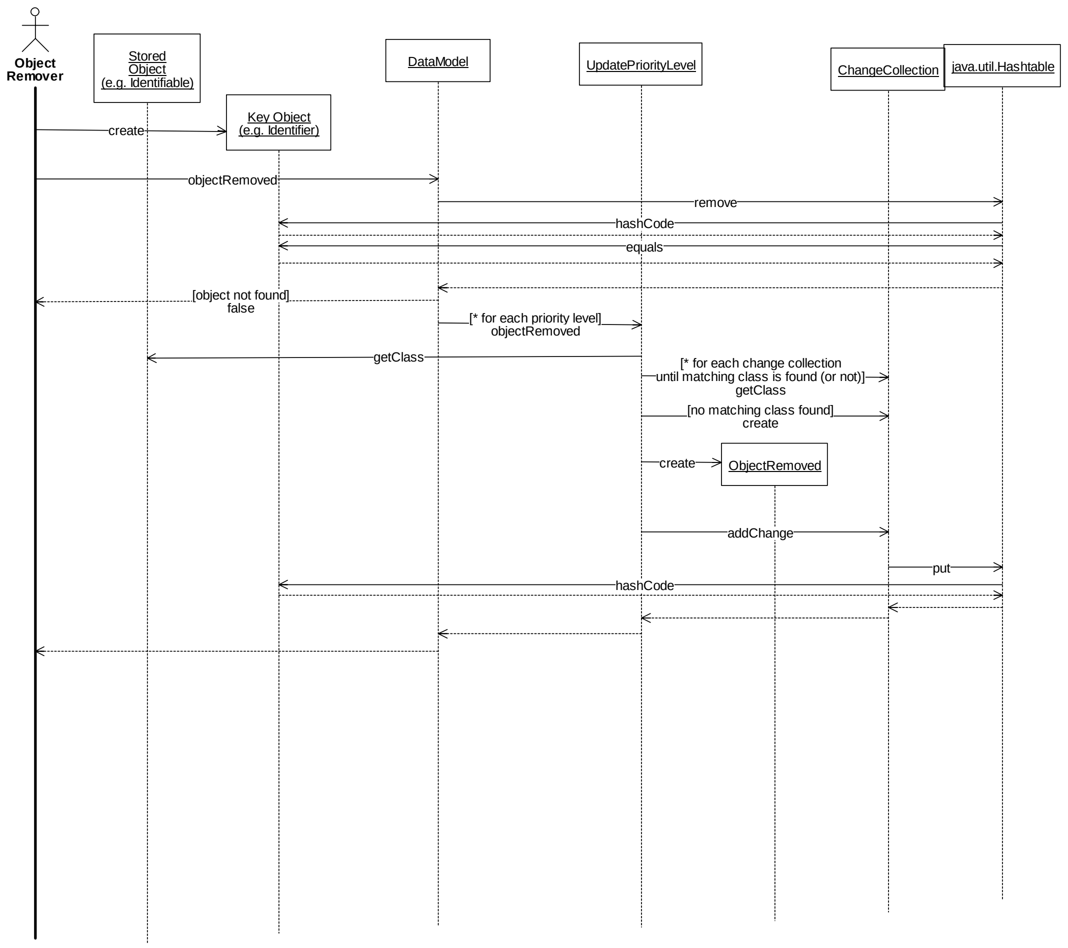

Value object to be added to the data model.
Key object to use for subsequent lookups of this value object.
Object Creator
DataModel
UpdatePriorityLevel
ChangeCollection
java.util.Hashtable
create
Object
create
Object (e.g. Identifier)
objectAdded
put
[* for each priority level] objectAdded
put
getClass
[* for each change collection until matching class is found (or not)] getClass
[no matching class found] create
create
ObjectAdded
addChange
put
previous value object with specified key

Figure 4. DataModel:ObjectAdded_ (Sequence Diagram)

3.2.2.3 DataModel:ObjectRemoved (Sequence Diagram)

This diagram shows what happens when an object is removed from the DataModel. The Key object is passed into the DataModel's objectRemoved method, which removes the stored object in the DataModel. If the object was removed (i.e., if it was found), the DataModel then calls the objectRemoved method for each UpdatePriorityLevel so that each priority level of observers will be updated independently. The UpdatePriorityLevel will check to see if it has a ChangeCollection to store changes for the class of the object. It will create a new ChangeCollection if necessary. The UpdatePriorityLevel will then create an ObjectRemoved object to represent the change. This object will be added to the ChangeCollection for the object's class. Java's garbage collection ensures that the object will not actually be deleted until the last reference to the object is removed; therefore, since object references are stored in the ChangeCollection objects, each object will exist at least until the last observer is updated on the lowest priority level. Observers have the responsibility to remove all of their references to the objects when their update method is called; otherwise, memory leaks will occur.



Object Remover
Stored Object (e.g. Identifiable)
DataModel
UpdatePriorityLevel
ChangeCollection
java.util.Hashtable
Key Object (e.g. Identifier)
create
objectRemoved
remove
hashCode
equals
[object not found] false
[* for each priority level] objectRemoved
getClass
[* for each change collection until matching class is found (or not)] getClass
[no matching class found] create
create
ObjectRemoved
addChange
put
hashCode

Figure 5. DataModel:ObjectRemoved (Sequence Diagram)

3.2.2.4 DataModel:ObjectUpdated (Sequence Diagram)

This diagram shows what happens when an object is updated through the DataModel. The caller passes in the Key object and an optional UpdateHint object. If an object is found with the Key, the DataModel will then call each UpdatePriorityLevel's objectUpdated method so that each priority level will be updated independently. The UpdatePriorityLevel checks to see if a ChangeCollection exists for the class of object that is being changed, and a ChangeCollection will be created if necessary. If there is a previous change for the object and the existing change is ObjectRemoved or ObjectAdded, the update will be ignored. Otherwise, the update hint will be combined with the existing update hints (if any) so that the resulting hints are a union of all hints which have been accumulated. The changes will be distributed to the observers when the next period expires for the UpdatePriorityLevel.

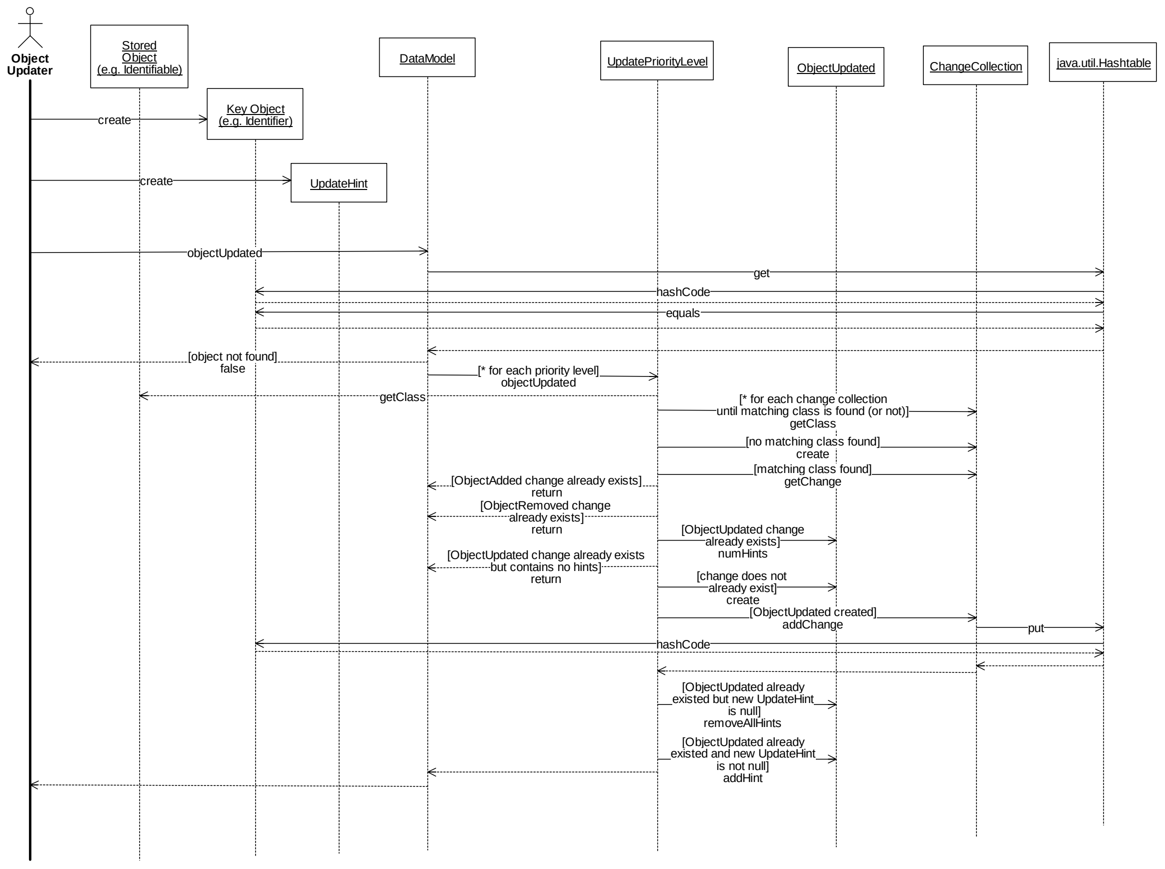

Object Updater
Stored Object (e.g. Identifiable)
DataModel
UpdatePriorityLevel
ObjectUpdated
ChangeCollection
java.util.Hashtable
create
Key Object (e.g. Identifier)
create
UpdateHint
objectUpdated
get
hashCode
equals
[object not found] false
[* for each priority level] objectUpdated
getClass
[* for each change collection until matching class is found (or not)] getClass
[no matching class found] create
[matching class found] getChange
[ObjectAdded change already exists] return
[ObjectRemoved change already exists] return
[ObjectUpdated change already exists] numHints
[ObjectUpdated change already exists but contains no hints] return
[change does not already exist] create
[ObjectUpdated created] addChange
put
hashCode
[ObjectUpdated already existed but new UpdateHint is null] removeAllHints
[ObjectUpdated already existed and new UpdateHint is not null] addHint

Figure 6. DataModel:ObjectUpdated (Sequence Diagram)

3.2.2.5 DataModel:UpdateObservers (Sequence Diagram)

This diagram shows how the observers are updated after changes have occurred to objects through the DataModel. The UpdatePriorityLevel thread decides that it's time to update the observers because the period has run out. It adds all of the changes that have been accumulated in the ChangeCollections and stores them in a ModelChange object. Then it distributes the ModelChange to all observers. If the observer is not a GUIObserver, it is updated on the UpdatePriorityLevel thread. However, GUIObservers must be updated on the main event thread, so the SwingUtility's invokeLater method is called to execute the update on the main event thread. After all observers are updated, the ChangeCollections are deleted to flush them. The UpdatePriorityLevel will then sleep until the next scheduled update.

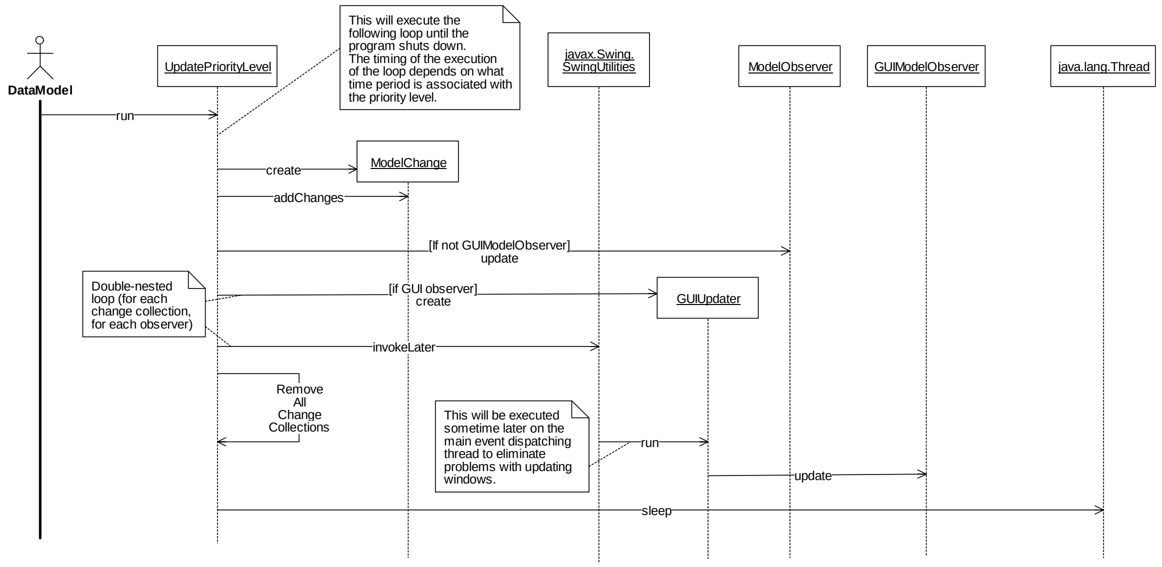

This will execute the following loop until the program shuts down. The timing of the execution of the loop depends on what time period is associated with the priority level.
DataModel
UpdatePriorityLevel
javax.Swing.SwingUtilities
ModelObserver
GUIModelObserver
java.lang.Thread
run
create
ModelChange
addChanges
[If not GUIModelObserver] update
Double-nested loop (for each change collection, for each observer)
[if GUI observer] create
GUIUpdater
invokeLater
Remove All Change Collections
This will be executed sometime later on the main event dispatching thread to eliminate problems with updating windows.
run
update
sleep

Figure 7. DataModel:UpdateObservers (Sequence Diagram)

3.3 DataTransfer

3.3.1 Class Diagrams

3.3.1.1 DataTransferClasses (Class Diagram)

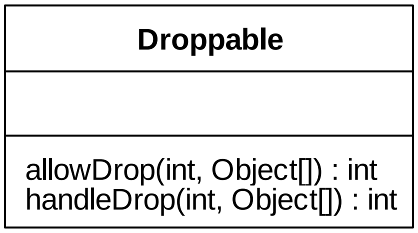

Droppable
allowDrop(int, Object[]) : int
handleDrop(int, Object[]) : int

Figure 8. DataTransferClasses (Class Diagram)

3.3.1.1.1 Droppable (Class)

This interface must be implemented by any object wishing to take part in a drag and drop operation. It is used by the DropHandler class to determine if a drop action should be allowed and to delegate the handling of the drop action after it is performed.

3.4 DMSUtility

3.4.1 Class Diagrams

3.4.1.1 DMSUtility (Class Diagram)

This Class Diagram shows classes related to the DMS that are used by both the GUI and the DMS service. Most of these classes are implementations of value type classes defined in the system interfaces (IDL).

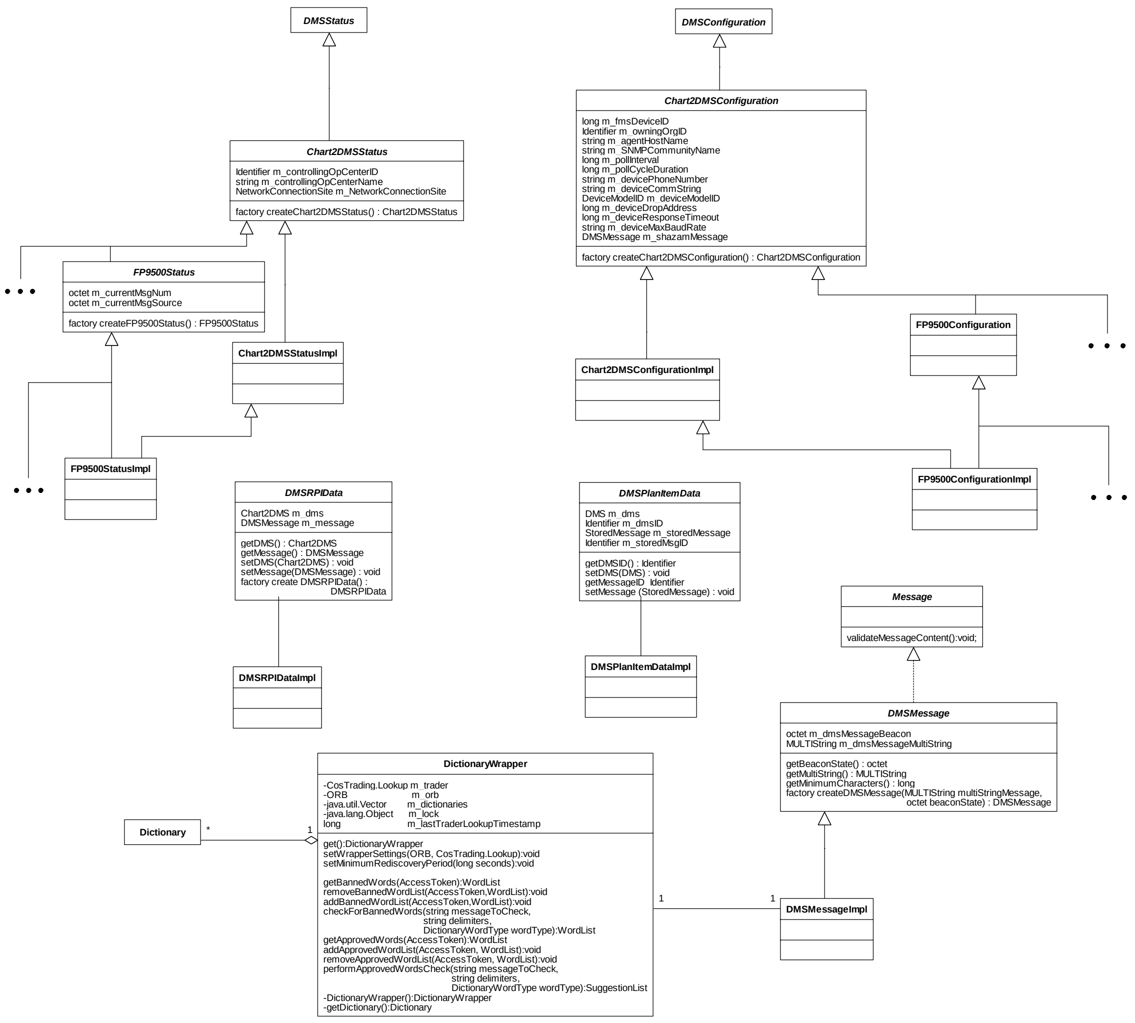

DMSStatus
DMSConfiguration
Chart2DMSConfiguration
long m_fmsDeviceID
Identifier m_owningOrgID
string m_agentHostName
string m_SNMPCommunityName
long m_pollInterval
long m_pollCycleDuration
string m_devicePhoneNumber
string m_deviceCommString
DeviceModelID m_deviceModelID
long m_deviceDropAddress
long m_deviceResponseTimeout
string m_deviceMaxBaudRate
DMSMessage m_shazamMessage
factory createChart2DMSConfiguration() : Chart2DMSConfiguration
Chart2DMSStatus
Identifier m_controllingOpCenterID
string m_controllingOpCenterName
NetworkConnectionSite m_NetworkConnectionSite
factory createChart2DMSStatus() : Chart2DMSStatus
FP9500Status
octet m_currentMsgNum
octet m_currentMsgSource
factory createFP9500Status() : FP9500Status
Chart2DMSStatusImpl
FP9500Configuration
Chart2DMSConfigurationImpl
FP9500StatusImpl
FP9500ConfigurationImpl
DMSRPIData
Chart2DMS m_dms
DMSMessage m_message
getDMS() : Chart2DMS
getMessage() : DMSMessage
setDMS(Chart2DMS) : void
setMessage(DMSMessage) : void
factory create DMSRPIData() : DMSRPIData
DMSPlanItemData
DMS m_dms
Identifier m_dmsID
StoredMessage m_storedMessage
Identifier m_storedMsgID
getDMSID() : Identifier
setDMS(DMS) : void
getMessageID Identifier
setMessage (StoredMessage) : void
Message
validateMessageContent():void;
DMSRPIDataImpl
DMSPlanItemDataImpl
DMSMessage
octet m_dmsMessageBeacon
MULTIString m_dmsMessageMultiString
getBeaconState() : octet
getMultiString() : MULTIString
getMinimumCharacters() : long
factory createDMSMessage(MULTIString multiStringMessage, octet beaconState) : DMSMessage
DictionaryWrapper
-CosTrading.Lookup m_trader
-ORB m_orb
-java.util.Vector m_dictionaries
-java.lang.Object m_lock
long m_lastTraderLookupTimestamp
get():DictionaryWrapper
setWrapperSettings(ORB, CosTrading.Lookup):void
setMinimumRediscoveryPeriod(long seconds):void
getBannedWords(AccessToken):WordList
removeBannedWordList(AccessToken,WordList):void
addBannedWordList(AccessToken,WordList):void
checkForBannedWords(string messageToCheck, string delimiters, DictionaryWordType wordType):WordList
getApprovedWords(AccessToken):WordList
addApprovedWordList(AccessToken, WordList):void
removeApprovedWordList(AccessToken, WordList):void
performApprovedWordsCheck(string messageToCheck, string delimiters, DictionaryWordType wordType):SuggestionList
-DictionaryWrapper():DictionaryWrapper
-getDictionary():Dictionary
Dictionary
*
1
1
1
DMSMessageImpl
• • •
• • •
• • •
• • •

Figure 9. DMSUtility (Class Diagram)

3.4.1.1.1 Chart2DMSConfiguration (Class)

The Chart2DMSConfiguration class is an abstract class which extends the DMSConfiguration class to provide configuration information specific to Chart II processing. Such information includes how to contact the sign under Chart II software control, the default SHAZAM message for using the sign as a HAR Notifier, and the owning organization. Such data extends beyond what would be industry-standard configuration information for a DMS.

3.4.1.1.2 Chart2DMSConfigurationImpl (Class)

The Chart2DMSConfigurationImpl class provides an implementation for the abstract Chart2DMSConfiguration class. It implements get and set methods to access and modify values of the configuration of a DMS. The configuration information stored here is normally fairly static: things like the size of the sign in characters and pixels, its name and location, and how to contact the sign (as opposed to dynamic information like the current message on the sign, which is stored in an analogous Status object).

3.4.1.1.3 Chart2DMSStatus (Class)

The Chart2DMSStatus class is an abstract class that extends the DMSStatus class to provide status information specific to Chart II processing, such as information on the controlling operations center for the sign. This data extends beyond what would be industry-standard status information for a DMS.

3.4.1.1.4 Chart2DMSStatusImpl (Class)

The Chart2DMSStatusImpl class provides an implementation for the abstract Chart2DMSStatus class. It implements get and set methods to access and modify values of the status of a DMS. The status information stored here is relatively dynamic: things like the current message on the sign, its beacon state, its current operational mode (online, offline, maintenance mode), and current operational status (OK, COMM_FAILURE, or HARDWARE_FAILURE) and controlling operations center. (More static information about the sign, such as its size and location, is stored in an analogous Configuration object.)

3.4.1.1.5 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerance by automatically failing over to a “working” dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.4.1.1.6 DMSMessage (Class)

The DMSMessage class is an abstract class which describes a message for a DMS. It consists of two elements: a MULTI-formatted message and beacon state information (whether the message requires that the beacons be on). The DMSMessage is contained within a DMSStatus object, used to communicate the current message on a sign, and is stored within a DMSRPIData object, used to specify the message that should be on a sign when the response plan item is executed.

3.4.1.1.7 DMSMessageImpl (Class)

The DMSMessageImpl class provides an implementation for the abstract DMSMessage class. It implements get and set methods to access and modify the MULTI-formatted message and beacon state values which make up a DMS message.

3.4.1.1.8 DMSPlanItemData (Class)

The DMSPlanItemData class is a valuetype that contains data stored in a plan item for a DMS. It is derived from PlanItemData.

3.4.1.1.9 DMSPlanItemDataImpl (Class)

The DMSPlanItemDataImpl class provides an implementation for the abstract DMSPlanItemData class. It implements get and set methods to access and modify values relative to a stored Plan Item for a DMS.

3.4.1.1.10 DMSRPIData (Class)

The DMSRPIData class is an abstract class that describes a response plan item for a DMS. It contains the unique identifier of the DMS to contain the DMSMessage, and the DMSMessage itself.

3.4.1.1.11 DMSRPIDataImpl (Class)

The DMSRPIDataImpl class provides an implementation for the abstract DMSRPIData class. It implements get and set methods to access and modify values relative to a Response Plan Item for a DMS.

3.4.1.1.12 FP9500Configuration (Class)

The FP9500Configuration class is an abstract class that extends the Chart2DMSConfiguration class to provide configuration information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.4.1.1.13 FP9500ConfigurationImpl (Class)

The FP9500ConfigurationImpl class provides an implementation for the abstract FP9500Configuration class. It implements get and set methods to access and modify values specific to the static configuration of an FP9500 DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.4.1.1.14 FP9500Status (Class)

The FP9500Status class is an abstract class that extends the Chart2DMSStatus class to provide status information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information. In this case, additional information provided the FP9500 model would include things like the current message number and current message source, status bits, light status, pixel failure map, and so on.

3.4.1.1.15 FP9500StatusImpl (Class)

The FP9500StatusImpl class provides an implementation for the abstract FP9500Status class. It implements get and set methods to access and modify values specific to the dynamic status configuration of an FP9500 DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific status information.

3.4.1.1.16 Message (Class)

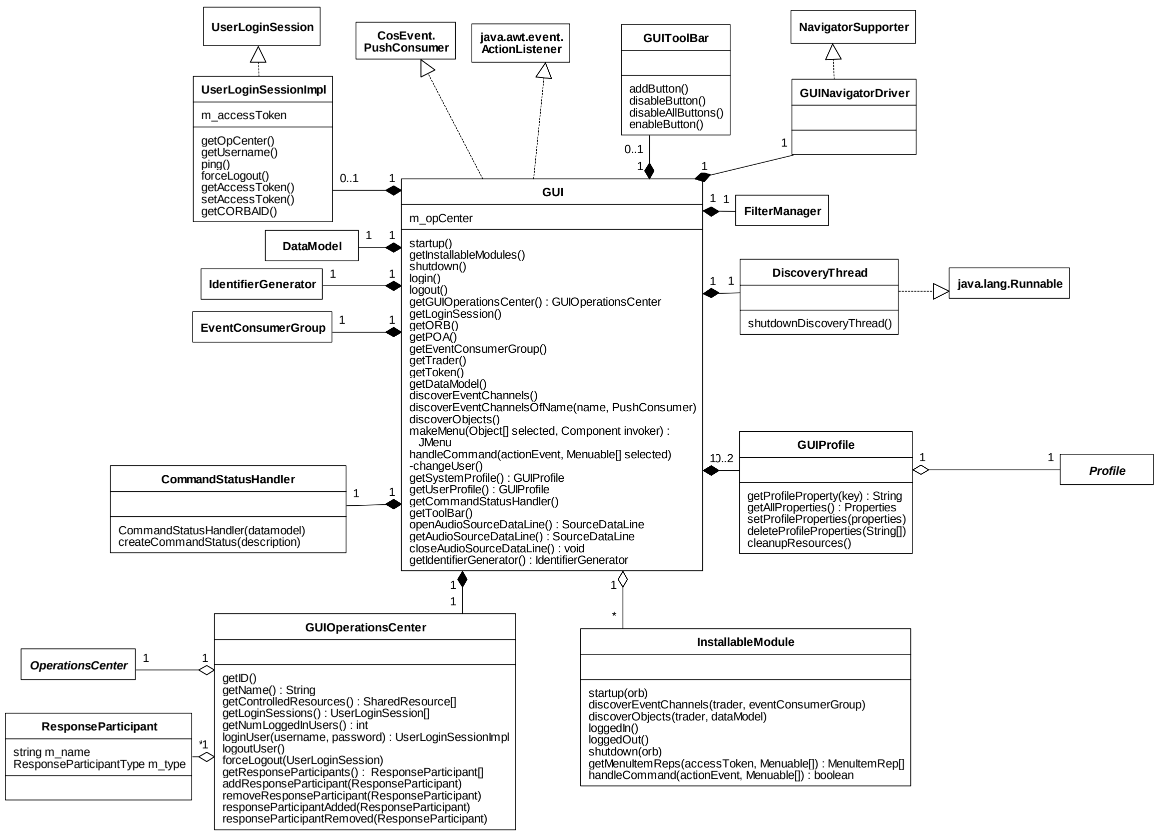
This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.5 GUI

3.5.1 Class Diagrams

3.5.1.1 R1B2GUIClassDiagram (Class Diagram)

This class diagram depicts the core classes and interfaces necessary to provide an extensible GUI application framework for future CHART II development. Included are details of objects served from the GUI application, an installable module framework, a core data model that provides the framework for window updates when objects change, and a framework for system preference configuration.



UserLoginSession
CosEvent.PushConsumer
java.awt.event.ActionListener
GUIToolBar
addButton()
disableButton()
disableAllButtons()
enableButton()
NavigatorSupporter
GUINavigatorDriver
UserLoginSessionImpl
m_accessToken
getOpCenter()
getUsername()
ping()
forceLogout()
getAccessToken()
setAccessToken()
getCORBAID()
GUI
m_opCenter
startup()
getInstallableModules()
shutdown()
login()
logout()
getGUIOperationsCenter() : GUIOperationsCenter
getLoginSession()
getORB()
getPOA()
getEventConsumerGroup()
getTrader()
getToken()
getDataModel()
discoverEventChannels()
discoverEventChannelsOfName(name, PushConsumer)
discoverObjects()
makeMenu(Object[] selected, Component invoker) : JMenu
handleCommand(actionEvent, Menuable[] selected)
-changeUser()
getSystemProfile() : GUIProfile
getUserProfile() : GUIProfile
getCommandStatusHandler()
getToolBar()
openAudioSourceDataLine() : SourceDataLine
getAudioSourceDataLine() : SourceDataLine
closeAudioSourceDataLine() : void
getIdentifierGenerator() : IdentifierGenerator
FilterManager
DataModel
IdentifierGenerator
EventConsumerGroup
DiscoveryThread
shutdownDiscoveryThread()
java.lang.Runnable
GUIProfile
getProfileProperty(key) : String
getAllProperties() : Properties
setProfileProperties(properties)
deleteProfileProperties(String[])
cleanupResources()
Profile
CommandStatusHandler
CommandStatusHandler(datamodel)
createCommandStatus(description)
GUIOperationsCenter
getID()
getName() : String
getControlledResources() : SharedResource[]
getLoginSessions() : UserLoginSession[]
getNumLoggedInUsers() : int
loginUser(username, password) : UserLoginSessionImpl
logoutUser()
forceLogout(UserLoginSession)
getResponseParticipants() : ResponseParticipant[]
addResponseParticipant(ResponseParticipant)
removeResponseParticipant(ResponseParticipant)
responseParticipantAdded(ResponseParticipant)
responseParticipantRemoved(ResponseParticipant)
InstallableModule
startup(orb)
discoverEventChannels(trader, eventConsumerGroup)
discoverObjects(trader, dataModel)
loggedIn()
loggedOut()
shutdown(orb)
getMenuItemReps(accessToken, Menuable[]) : MenuItemRep[]
handleCommand(actionEvent, Menuable[]) : boolean
OperationsCenter
ResponseParticipant
string m_name
ResponseParticipantType m_type
0..1
1
1..2
*

Figure 11. R1B2GUIClassDiagram (Class Diagram)

3.5.1.1.1 CommandStatusHandler (Class)

This class provides functionality that allows the modules to deal with CommandStatus objects for calling asynchronous methods without performing the housekeeping associated with serving these objects. It provides a method for creating a CommandStatus object which will create the object, attach it to the ORB, add it to the data model, and observe the data model waiting for the CommandStatus object to complete. When it completes, this object will disconnect it from the ORB and remove it from the data model.

3.5.1.1.2 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.5.1.1.3 EventConsumerGroup (Class)

This class represents a collection of event consumers that will be monitored to verify that they do not lose their connection to the CORBA event service. The class will periodically ask each consumer to verify its connection to the event channel on which it is dependent to receive events.

3.5.1.1.4 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.5.1.1.5 DiscoveryThread (Class)

This thread is used by the GUI to check for new event channels and served CORBA objects. It will periodically call the GUI to find event channels and objects.

3.5.1.1.6 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.5.1.1.7 GUINavigatorDriver (Class)

This class handles all of the Navigator-specific functionality for the GUI.

3.5.1.1.8 GUIOperationsCenter (Class)

This class is a GUI “wrapper” object that is used to wrap an OperationsCenter object. The wrapping is done to cache the data locally for faster access, and to provide GUI-specific functionalities to the wrapped object.

3.5.1.1.9 GUIToolBar (Class)

This class will hold all of the top-level buttons and will be the launching point for invoking the functionality of the CHART2 system. It will be created at startup, and each module may add any toolbar buttons at that time. At Login, modules that have added toolbar buttons at startup should enable any toolbar buttons that should be enabled (depending on access rights). The buttons will be disabled by the GUI after they are added at startup and again at logout.

3.5.1.1.10 FilterManager (Class)

This class provides functionality for managing the filters in the system. As it deals with the singleton GUI and the DataModel objects, it too will be a singleton object. The GUI will create and hold the FilterManager. Filter supporters can be added to the FilterManager to support the creation of supporter-specific filter types.

3.5.1.1.11 GUIProfile (Class)

This class is a wrapper for the Profile CORBA interface. It provides GUI-specific functionality for the profile. A GUIProfile can represent either a system profile or a user profile.

3.5.1.1.12 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.5.1.1.13 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.5.1.1.14 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.5.1.1.15 NavigatorSupporter (Class)

This interface must be implemented by any subsystem that supports invoking the Navigator. It must be able to supply the Navigable objects, and also can support user interaction with the selected Navigable objects through menus and drag/drop.

3.5.1.1.16 ResponseParticipant (Class)

The ResponseParticipant class is a non-behavioral structure that specifies a participant in a response.

3.5.1.1.17 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

3.5.1.1.18 Profile (Class)

This class contains a set of user or administrator defined properties that are used to configure how the CHART II system behaves or presents information to a user.

3.5.1.1.19 IdentifierGenerator (Class)

This class is used to create and manipulate identifiers that are to be used in Identifiable objects.

3.5.1.1.20 UserLoginSession (Class)

The UserLoginSession CORBA interface is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

3.5.1.1.21 UserLoginSessionImpl (Class)

This class is the implementation of the CORBA UserLoginSession interface. It will be served from the GUI and will be passed to the OperationsCenter on login. It will also store the access token returned from the OperationsCenter.

3.5.1.2 MiscClasses (Class Diagram)

This diagram shows other classes that are used in the GUI, but are not part of the fundamental framework of the GUI.

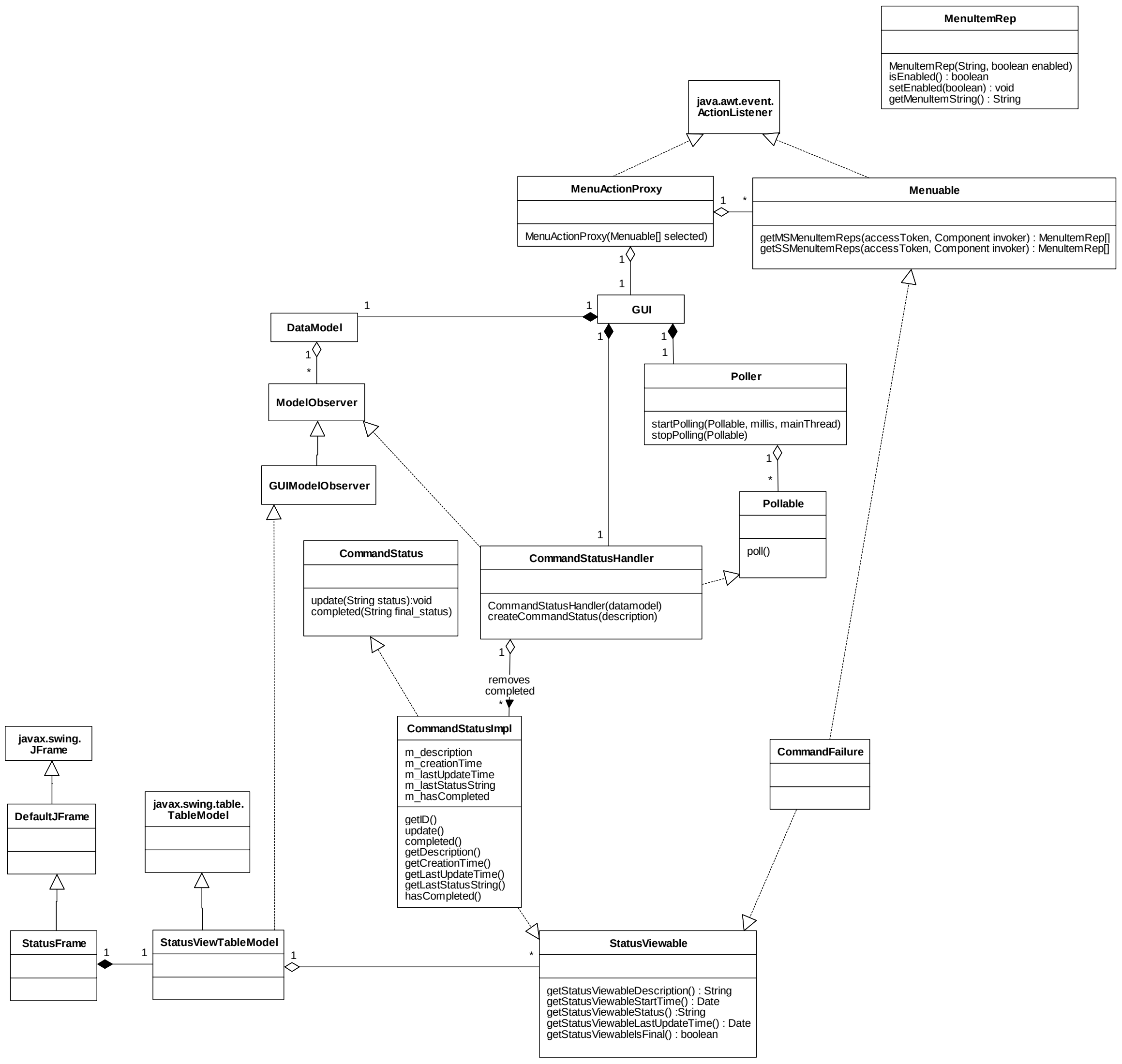

MenuItemRep
MenuItemRep(String, boolean enabled)
isEnabled() : boolean
setEnabled(boolean) : void
getMenuItemString() : String
java.awt.event.ActionListener
MenuActionProxy
MenuActionProxy(Menuable[] selected)
Menuable
getMSMenuItemReps(accessToken, Component invoker) : MenuItemRep[]
getSSMenuItemReps(accessToken, Component invoker) : MenuItemRep[]
GUI
DataModel
ModelObserver
GUIModelObserver
Poller
startPolling(Pollable, millis, mainThread)
stopPolling(Pollable)
Pollable
poll()
CommandStatus
update(String status):void
completed(String final_status)
CommandStatusHandler
CommandStatusHandler(datamodel)
createCommandStatus(description)
removes completed
CommandStatusImpl
m_description
m_creationTime
m_lastUpdateTime
m_lastStatusString
m_hasCompleted
getID()
update()
completed()
getDescription()
getCreationTime()
getLastUpdateTime()
getLastStatusString()
hasCompleted()
CommandFailure
javax.swing.JFrame
DefaultJFrame
javax.swing.table.TableModel
StatusFrame
StatusViewTableModel
StatusViewable
getStatusViewableDescription() : String
getStatusViewableStartTime() : Date
getStatusViewableStatus() :String
getStatusViewableLastUpdateTime() : Date
getStatusViewableIsFinal() : boolean

Figure 12. MiscClasses (Class Diagram)

3.5.1.2.1 CommandFailure (Class)

This object represents a failure of a command. It implements the StatusViewable interface so that it can be displayed in the StatusFrame (i.e., the “Command Failures” window).

3.5.1.2.2 CommandStatus (Class)

The CommandStatus CORBA interface is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

3.5.1.2.3 CommandStatusImpl (Class)

This class is the implementation of the CommandStatus CORBA interface. It will be created and passed to a server when a command is to be executed so that the GUI can stay updated as the command is executing.

3.5.1.2.4 CommandStatusHandler (Class)

This class provides functionality that allows the modules to deal with CommandStatus objects for calling asynchronous methods without performing the housekeeping associated with serving these objects. It provides a method for creating a CommandStatus object which will create the object, attach it to the ORB, add it to the data model, and observe the data model waiting for the CommandStatus object to complete. When it completes, this object will disconnect it from the ORB and remove it from the data model.

3.5.1.2.5 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects which are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.5.1.2.6 DefaultJFrame (Class)

This class provides a default implementation of the WindowManageable interface, and may be used as a base class for other frame windows in the GUI. It handles all interactions with the WindowManager for attaching and detaching, as well as saving the window position.

3.5.1.2.7 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.5.1.2.8 ModelObserver (Class)

This interface must be implemented by any object that may need to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

3.5.1.2.9 GUIModelObserver (Class)

Interface to be implemented by GUI components that may needd to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

3.5.1.2.10 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.5.1.2.11 javax.swing. JFrame (Class)

Java class that displays a frame window.

3.5.1.2.12 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user’s access rights before supplying the strings, so the user’s actions may be restricted.

3.5.1.2.13 MenuActionProxy (Class)

This class catches the action performed by the menu item and stores the selected menuable objects to act on.

3.5.1.2.14 MenuItemRep (Class)

This class is used by the Menuable objects when they are called to return their menu items. It contains a flag indicating whether the menu item is to be disabled.

3.5.1.2.15 javax.swing.table. TableModel (Class)

This class provides the data structure that drives the population and updating of the data used by the JTable (a Java GUI component).

3.5.1.2.16 StatusViewable (Class)

This interface provides the functionality needed to add objects to the StatusViewTableModel so that they can be displayed in the StatusFrame.

3.5.1.2.17 StatusViewTableModel (Class)

This class provides the data framework needed to populate and update the JTable that displays the StatusViewable objects in the StatusFrame.

3.5.1.2.18 Pollable (Class)

This interface provides a method so that the Poller can periodically call the object to poll it.

3.5.1.2.19 Poller (Class)

This class will periodically call the poll() method for any Pollables for which polling has been started. This happens either on the polling thread or on the AWT event thread, as specified when the polling is started.

3.5.1.2.20 StatusFrame (Class)

This class is a window that displays the StatusViewable objects in a JTable. Currently the “Command Status” and “Command Failures” windows are StatusFrames.

3.5.2 Sequence Diagrams

3.5.2.1 GUI:ChangeUserBasic (Sequence Diagram)

This diagram shows the steps that will be taken in the GUI when a user change occurs without first logging out. The new user will be logged in and the previous user will be logged out, then all windows are closed and the new user's preferences are loaded to replace the previous preferences. If the changeUser command fails, the previous user will still be logged in and the new user will not be logged in.

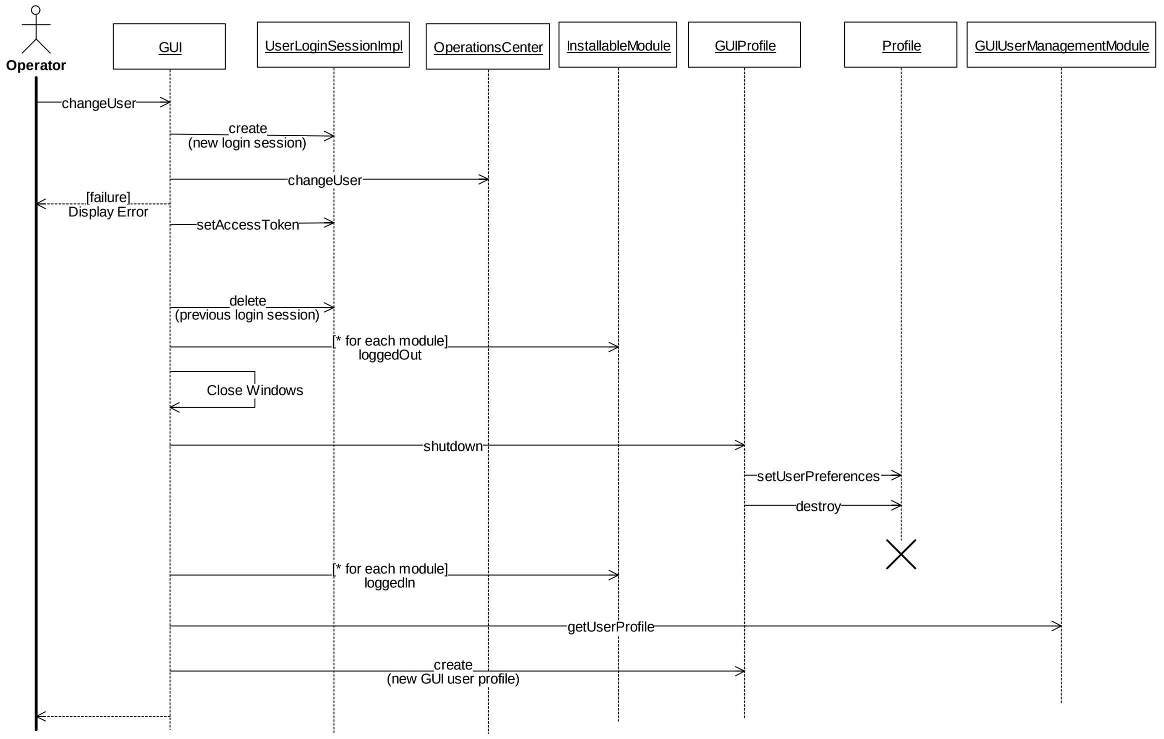
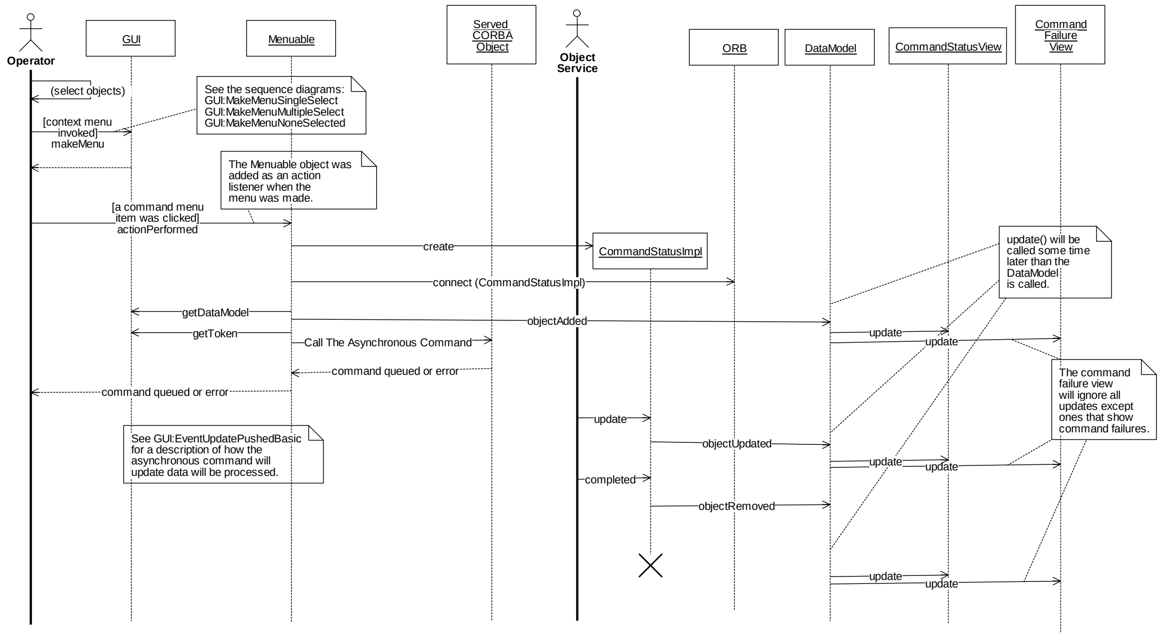


Figure 13. GUI:ChangeUserBasic (Sequence Diagram)

3.5.2.2 GUI:CommandObjectBasic (Sequence Diagram)

This diagram shows the basic steps involved in issuing a typical command to an object. The context menu is built when the user right clicks on one or more selected objects. At this time the GUI wrapper object will be added as an ActionListener and will receive the command if any of its menu items are clicked on. (See the sequence diagrams GUI:MakeMenuSingleSelect and GUI:MakeMenuMultipleSelect for more details). If a long-running command is invoked, the object will create a CommandStatusImpl object, put it in the DataModel, and pass it to the server so that the server can call back as the command is completed. When the server calls the CommandStatusImpl's completed() method, the CommandStatusImpl will remove itself from the DataModel. If the command fails, the Command Failures window will add the command status to its displayed list.



Operator
GUI
Menuable
Served CORBA Object
Object Service
ORB
DataModel
CommandStatusView
Command Failure View
(select objects)
See the sequence diagrams: GUI:MakeMenuSingleSelect GUI:MakeMenuMultipleSelect GUI:MakeMenuNoneSelected
[context menu invoked] makeMenu
The Menuable object was added as an action listener when the menu was made.
[a command menu item was clicked] actionPerformed
create
CommandStatusImpl
update() will be called some time later than the DataModel is called.
connect (CommandStatusImpl)
getDataModel
objectAdded
getToken
update
update
Call The Asynchronous Command
command queued or error
command queued or error
The command failure view will ignore all updates except ones that show command failures.
update
See GUI:EventUpdatePushedBasic for a description of how the asynchronous command will update data will be processed.
objectUpdated
update
update
completed
objectRemoved
update
update

Figure 14. GUI:CommandObjectBasic (Sequence Diagram)

3.5.2.3 GUI:DiscoveryBasic (Sequence Diagram)

This diagram shows the ongoing discovery of event channels and served CORBA objects. In the GUI's startup, it will start the DiscoveryThread, which will periodically search for new event channels and objects until the GUI shuts down. The event channels are discovered before the objects to prevent the dropping of events just after the objects are discovered. First, the GUI looks for resource watchdog event channels, which will inform the user if resources are controlled by an Operations Center which does not have any logged in users, then it asks the modules to look for the module-specific event channels. If any event channels are found, they are added to the EventConsumerGroup, which will maintain the connection to the event channel if the event service goes down and is restarted. The GUI will then ask each module to discover the objects that it is interested in. Each module will look up the object factory in the trader, and ask the object factory for all of its objects. Then, the module will check whether the CORBA object already has a GUI wrapper object stored in the DataModel. If it doesn't, it will create a new wrapper object and add it to the DataModel. Any ModelObservers that are attached to the DataModel will be subsequently informed of the new wrapper objects.

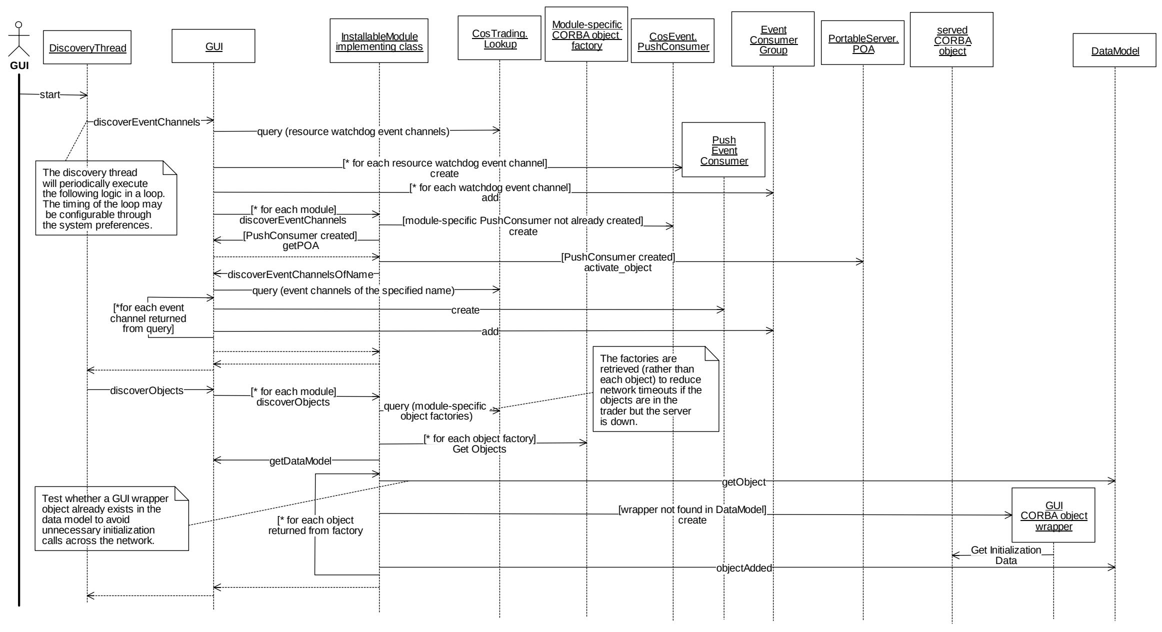

GUI
DiscoveryThread
GUI
InstallableModule implementing class
CosTrading. Lookup
Module-specific CORBA object factory
CosEvent. PushConsumer
Event Consumer Group
PortableServer. POA
served CORBA object
DataModel
start
discoverEventChannels
query (resource watchdog event channels)
Push Event Consumer
[* for each resource watchdog event channel] create
The discovery thread will periodically execute the following logic in a loop. The timing of the loop may be configurable through the system preferences.
[* for each watchdog event channel] add
[* for each module] discoverEventChannels
[module-specific PushConsumer not already created] create
[PushConsumer created] getPOA
[PushConsumer created] activate_object
discoverEventChannelsOfName
query (event channels of the specified name)
[*for each event channel returned from query]
create
add
The factories are retrieved (rather than each object) to reduce network timeouts if the objects are in the trader but the server is down.
discoverObjects
[* for each module] discoverObjects
query (module-specific object factories)
[* for each object factory] Get Objects
getDataModel
getObject
Test whether a GUI wrapper object already exists in the data model to avoid unnecessary initialization calls across the network.
[* for each object returned from factory
[wrapper not found in DataModel] create
GUI CORBA object wrapper
Get Initialization Data
objectAdded

Figure 15. GUI:DiscoveryBasic (Sequence Diagram)

3.5.2.4 GUI:EventUpdatePushedBasic (Sequence Diagram)

This diagram shows how updates to the served CORBA objects propagate to the GUI windows. The server will push the event data to the event service. The CORBA event service will then push the event data to the PushConsumer (which would typically be the GUI or an InstallableModule). The event data must contain some identification data so that the GUI wrapper object can be looked up in the DataModel. After the PushConsumer retrieves the GUI wrapper object from the DataModel, it will update any relevant data within the object and will call the DataModel one or more times with update hints to indicate what part of the object's data changed. The DataModel will accumulate all of the update hints for some short time period until it distributes them to all of the attached ModelObservers (which would typically be windows displaying the object data).

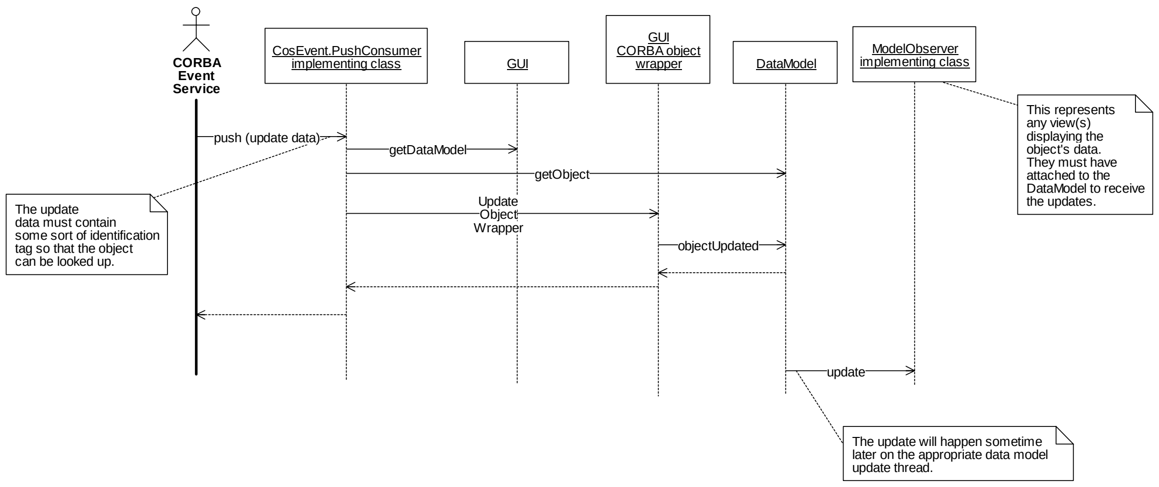

CORBA Event Service
CosEvent.PushConsumer implementing class
GUI
GUI CORBA object wrapper
DataModel
ModelObserver implementing class
This represents any view(s) displaying the object's data. They must have attached to the DataModel to receive the updates.
push (update data)
getDataModel
getObject
Update Object Wrapper
objectUpdated
The update data must contain some sort of identification tag so that the object can be looked up.
update
The update will happen sometime later on the appropriate data model update thread.

Figure 16. GUI:EventUpdatePushedBasic (Sequence Diagram)

3.5.2.5 GUI:LoginBasic (Sequence Diagram)

This diagram shows what steps must be taken at login. The GUI creates a UserLoginSessionImpl and passes it to the OperationsCenter for login. The GUI will then store the AccessToken in the UserLoginSessionImpl for later use. The GUI then enables the basic buttons on the GUI toolbar. Then it creates the GUIProfile wrappers for the system and user Profiles, and it initializes the system and user Navigator filters. Then the GUI calls each InstallableModule to allow them to handle post-login processing.

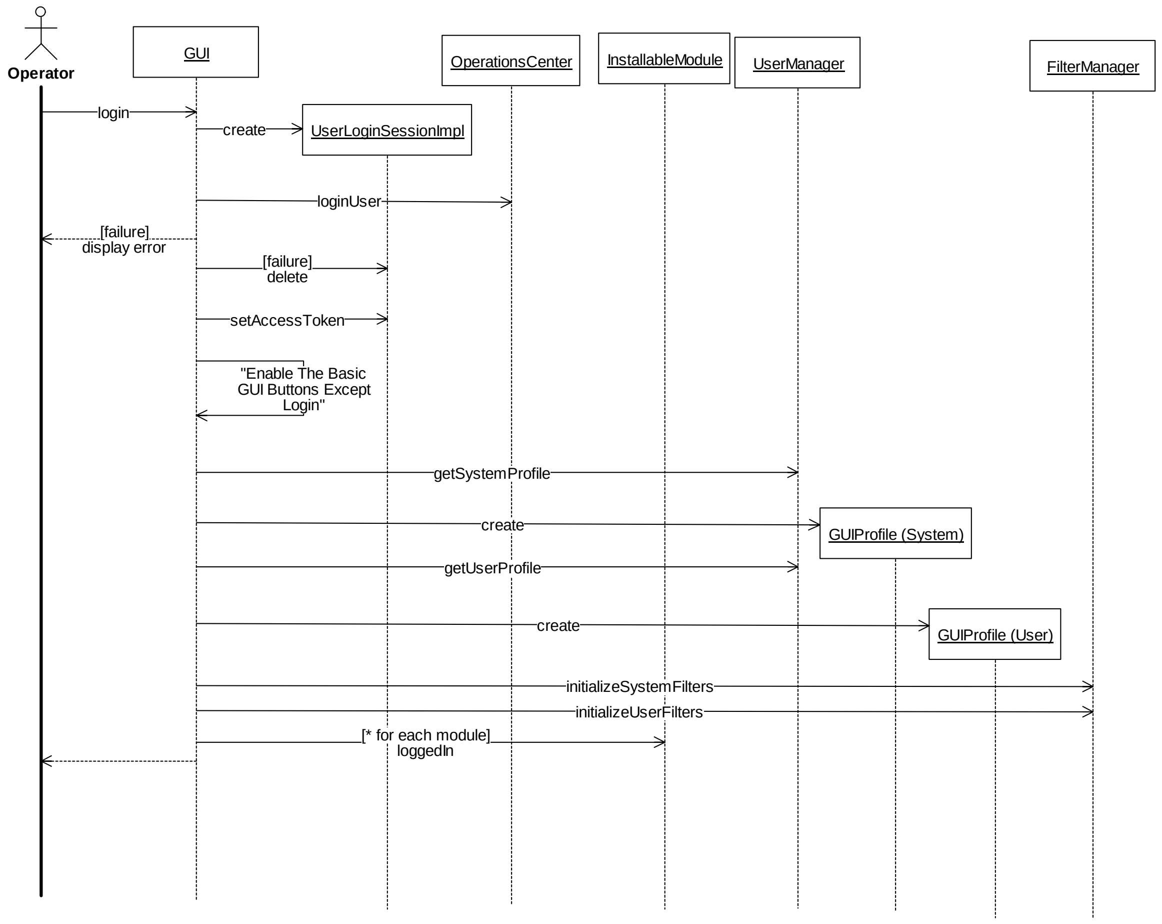


Figure 17. GUI:LoginBasic (Sequence Diagram)

3.5.2.6 GUI:LogoutBasic (Sequence Diagram)

This diagram shows what processing happens when the user logs out. The GUI calls the GUIOperationsCenter, which in turn calls the OperationsCenter object. If any shared resources are still assigned to the Op Center and the user logging out is the last user at the Op Center, the logout will fail and the user will need to transfer the shared resources to another Op Center. In this case a dialog will be displayed. If the logout is successful, the GUI will call each installable module's loggedOut() method. Then it will close all windows and disable the toolbar buttons, deactivate the UserLoginSessionImpl, and clean up the system and user Navigator filters and GUIProfile objects.

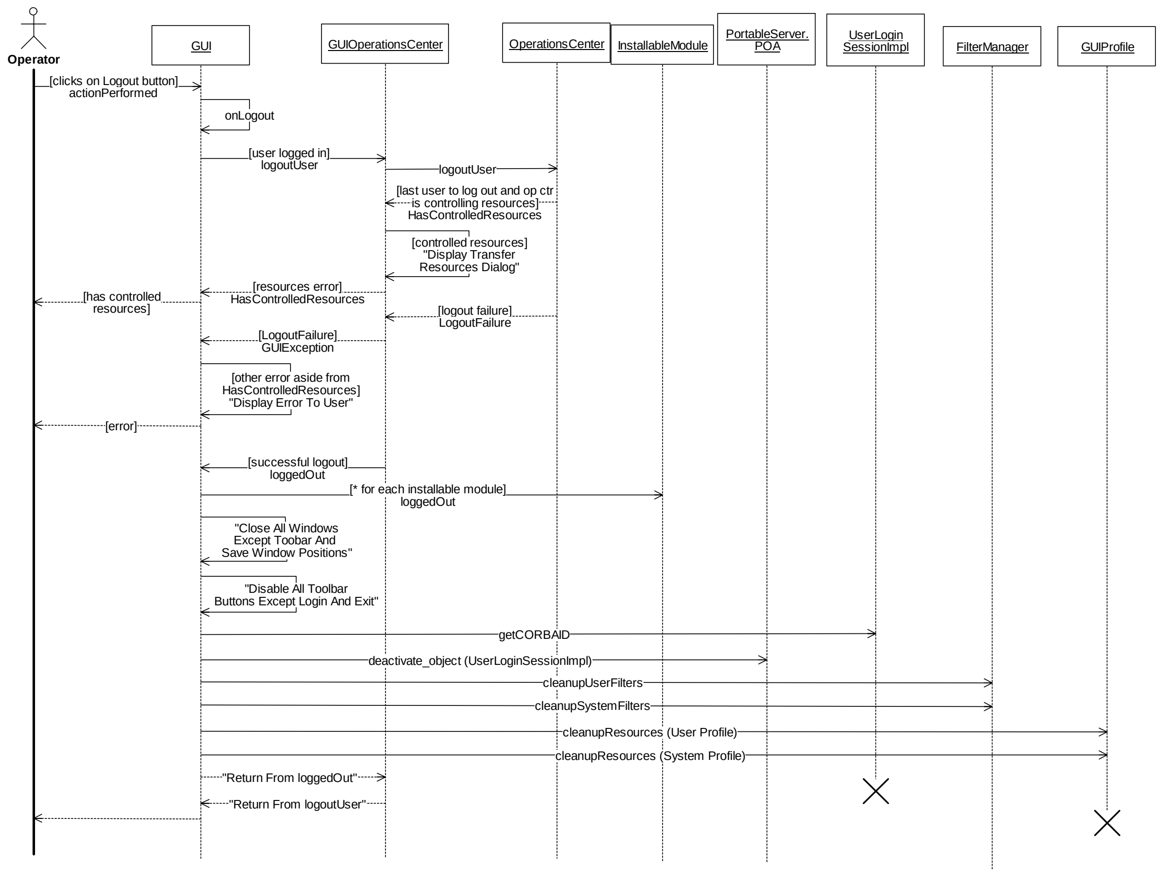

Operator
GUI
GUIOperationsCenter
OperationsCenter
InstallableModule
PortableServer.POA
UserLogin SessionImpl
FilterManager
GUIProfile
[clicks on Logout button] actionPerformed
onLogout
[user logged in] logoutUser
logoutUser
[last user to log out and op ctr is controlling resources] HasControlledResources
[controlled resources] "Display Transfer Resources Dialog"
[resources error] HasControlledResources
[has controlled resources]
[logout failure] LogoutFailure
[LogoutFailure] GUIException
[other error aside from HasControlledResources] "Display Error To User"
[error]
[successful logout] loggedOut
[* for each installable module] loggedOut
"Close All Windows Except Toobar And Save Window Positions"
"Disable All Toolbar Buttons Except Login And Exit"
getCORBAID
deactivate_object (UserLoginSessionImpl)
cleanupUserFilters
cleanupSystemFilters
cleanupResources (User Profile)
cleanupResources (System Profile)
"Return From loggedOut"
"Return From logoutUser"

Figure 18. GUI:LogoutBasic (Sequence Diagram)

3.5.2.7 GUI:MakeMenuMultipleSelect (Sequence Diagram)

This diagram shows how a menu is created when two or more GUI wrapper objects are selected. The GUI's makeMenu method determines that there are multiple objects selected, and it creates a BucketSet that it will use to count the menu items. Then it asks each selected object to supply the multiple-selection menu item reps. If the user does not have sufficient rights, those menu items will be grayed out. The menu item strings are put into the BucketSet and then retrieved. The only reps that are retrieved from the BucketSet are those which have the same number of instances as there are selected objects. The GUI then creates menu items for the reps and attaches a new MenuActionProxy as an ActionListener to each representative menu item.

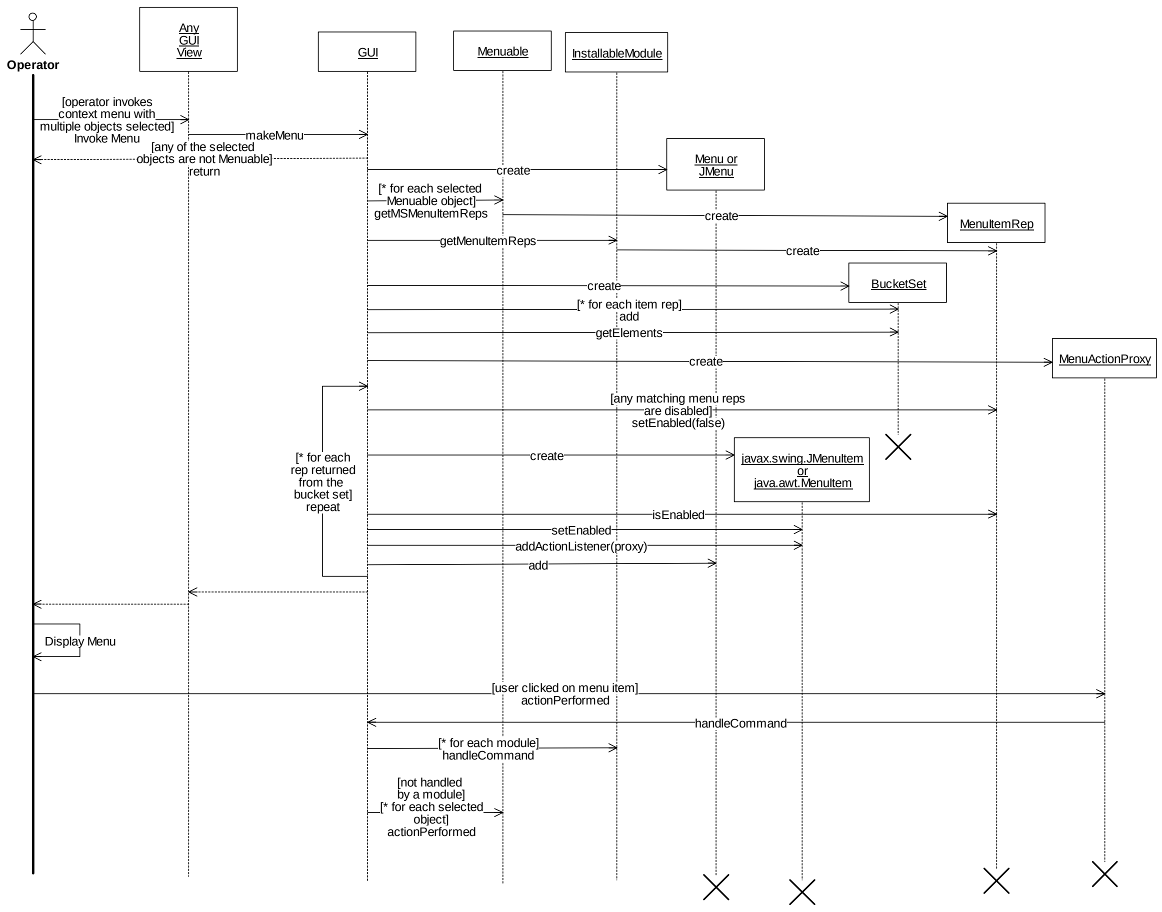

Operator
Any GUI View
GUI
Menuable
InstallableModule
[operator invokes context menu with multiple objects selected] Invoke Menu
makeMenu
[any of the selected objects are not Menuable] return
create
Menu or JMenu
[* for each selected Menuable object] getMSMenuItemReps
create
MenuItemRep
getMenuItemReps
create
create
BucketSet
[* for each item rep] add
getElements
create
MenuActionProxy
[any matching menu reps are disabled] setEnabled(false)
[* for each rep returned from the bucket set] repeat
create
javax.swing.JMenuItem or java.awt.MenuItem
isEnabled
setEnabled
addActionListener(proxy)
add
Display Menu
[user clicked on menu item] actionPerformed
handleCommand
[* for each module] handleCommand
[not handled by a module] [* for each selected object] actionPerformed

Figure 19. GUI:MakeMenuMultipleSelect (Sequence Diagram)

3.5.2.8 GUI:MakeMenuNoneSelected (Sequence Diagram)

This diagram shows how a menu is created when no GUI wrapper objects are selected. The GUI's makeMenu method determines that there are no objects selected, and the GUI then adds its own global menu items and calls each module to get their menu item reps. The GUI then creates a MenuActionProxy and attaches it as an ActionListener to the menu items so that it will be called when the user clicks on the menu items. If the user does not have rights to perform the action associated with a menu item, it will be grayed out.

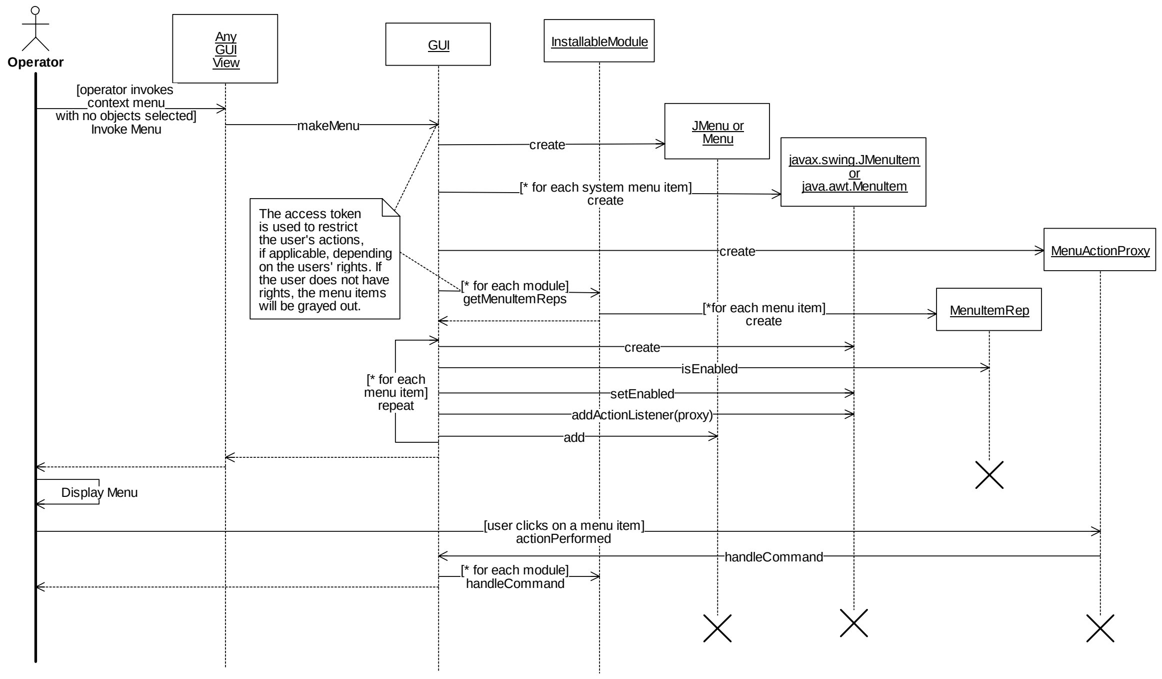

Operator
Any GUI View
GUI
InstallableModule
[operator invokes context menu with no objects selected] Invoke Menu
makeMenu
create
JMenu or Menu
javax.swing.JMenuItem or java.awt.MenuItem
[* for each system menu item] create
The access token is used to restrict the user's actions, if applicable, depending on the users' rights. If the user does not have rights, the menu items will be grayed out.
create
MenuActionProxy
[* for each module] getMenuItemReps
[*for each menu item] create
MenuItemRep
create
isEnabled
[* for each menu item] repeat
setEnabled
addActionListener(proxy)
add
Display Menu
[user clicks on a menu item] actionPerformed
handleCommand
[* for each module] handleCommand

Figure 20. GUI:MakeMenuNoneSelected (Sequence Diagram)

3.5.2.9 GUI:MakeMenuSingleSelect (Sequence Diagram)

This diagram shows how a menu is created when exactly one GUI wrapper object is selected. The GUI's makeMenu method determines that there is one object selected, and it asks the Menuable object for the single-select menu item strings. The GUI will then create all of the menu items and attach a new MenuActionProxy as the ActionListener to each of the menu items.

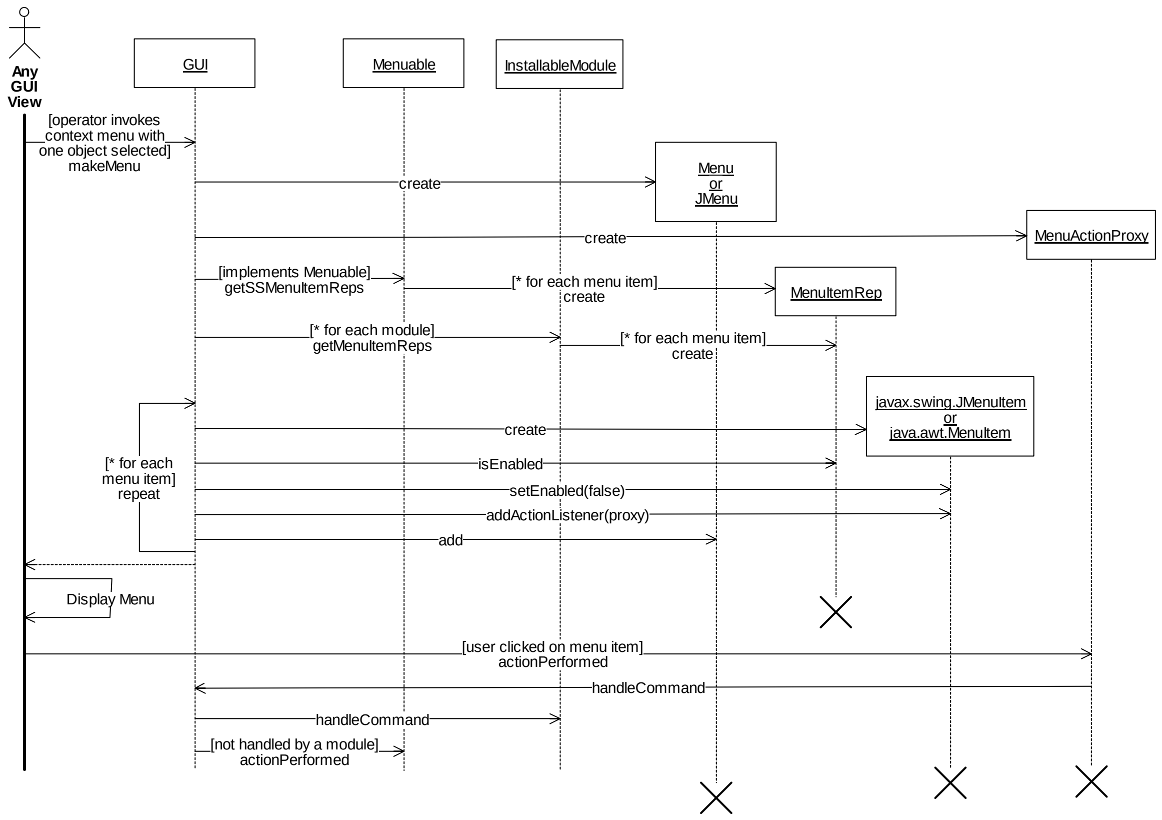

Any GUI View
GUI
Menuable
InstallableModule
[operator invokes context menu with one object selected] makeMenu
create
Menu or JMenu
create
MenuActionProxy
[implements Menuable] getSSMenuItemReps
[* for each menu item] create
MenuItemRep
[* for each module] getMenuItemReps
[* for each menu item] create
javax.swing.JMenuItem or java.awt.MenuItem
create
[* for each menu item] repeat
isEnabled
setEnabled(false)
addActionListener(proxy)
add
Display Menu
[user clicked on menu item] actionPerformed
handleCommand
handleCommand
[not handled by a module] actionPerformed

Figure 21. GUI:MakeMenuSingleSelect (Sequence Diagram)

3.5.2.10 GUI:ShutdownBasic (Sequence Diagram)

This diagram shows steps necessary for a shutdown. The operator either closes the GUIToolBar or clicks on the Exit button. Either of these actions will result in the GUI's shutdown method being called. If the user is logged in, he or she will be logged out. If this happens, the GUI calls the GUIOperationsCenter, which in turn calls the OperationsCenter object. If any shared resources are still assigned to the Op Center and the user logging out is the last user at the Op Center, the logout will fail and the user will need to transfer the shared resources to another Op Center. In this case a dialog will be displayed. If the logout is successful, the GUI will call each installable module's loggedOut() method. Then it will close all windows and disable the toolbar buttons, deactivate the UserLoginSessionImpl, and clean up the system and user Navigator filters and GUIProfile objects. Once the user is logged out, the GUI shuts down the discovery thread and informs all of the modules that the GUI is being shut down. Finally, the GUI process exits.

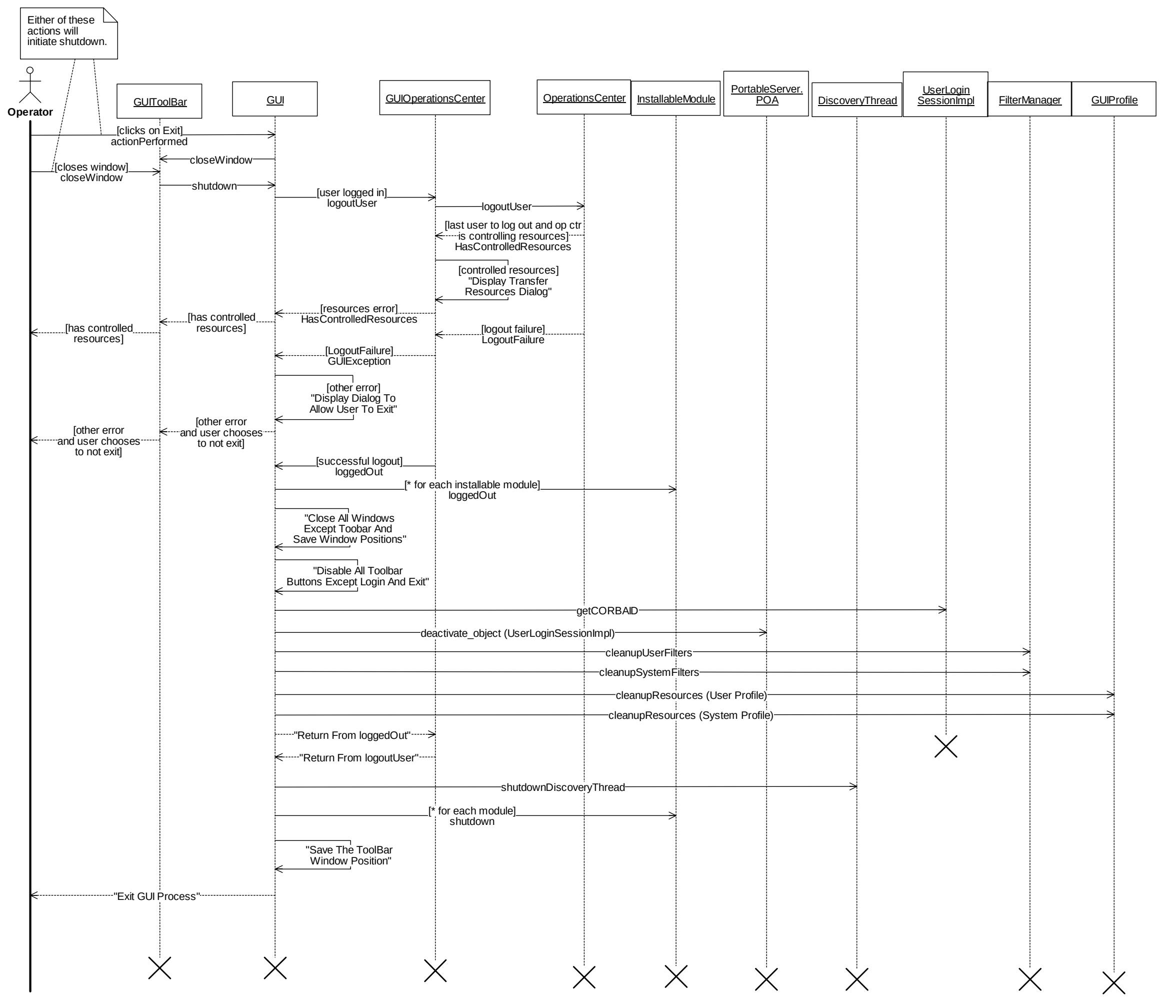

Either of these actions will initiate shutdown.
Operator
GUIToolBar
GUI
GUIOperationsCenter
OperationsCenter
InstallableModule
PortableServer.POA
DiscoveryThread
UserLogin SessionImpl
FilterManager
GUIProfile
[clicks on Exit] actionPerformed
closeWindow
[closes window] closeWindow
shutdown
[user logged in] logoutUser
logoutUser
[last user to log out and op ctr is controlling resources] HasControlledResources
[controlled resources] "Display Transfer Resources Dialog"
[resources error] HasControlledResources
[has controlled resources]
[has controlled resources]
[logout failure] LogoutFailure
[LogoutFailure] GUIException
[other error] "Display Dialog To Allow User To Exit"
[other error and user chooses to not exit]
[other error and user chooses to not exit]
[successful logout] loggedOut
[* for each installable module] loggedOut
"Close All Windows Except Toobar And Save Window Positions"
"Disable All Toolbar Buttons Except Login And Exit"
getCORBAID
deactivate_object (UserLoginSessionImpl)
cleanupUserFilters
cleanupSystemFilters
cleanupResources (User Profile)
cleanupResources (System Profile)
"Return From loggedOut"
"Return From logoutUser"
shutdownDiscoveryThread
[* for each module] shutdown
"Save The ToolBar Window Position"
"Exit GUI Process"

Figure 22. GUI:ShutdownBasic (Sequence Diagram)

3.5.2.11 GUI:StartupBasic (Sequence Diagram)

When the GUI application is started, it first performs CORBA initialization: it initializes the CORBA ORB and creates the root POA and the persistent POA, and activates the POA Manager. Then it creates a GUIStartupCommand object, which is passed off to Java to execute at a later time on Java's AWT event thread. The ORB's run() method is called, which blocks the application's main thread. Java then invokes the GUIStartupCommand, which creates and initializes the GUI. During startup(), the GUI is activated in the POA so that it can receive CORBA events for the ResourceManagement events. It also loads the names of the InstallableModules to install from the system properties file, and proceeds to instantiate all of the installable modules based on the class name of each module. The GUI then looks for the CORBA Trading Service, and queries the OperationsCenter object with the name specified in the system properties file. It also queries the UserManager object from the Trading Service. If all of this is successful, the GUI's toolbar is created and the buttons are added. Each module's startup() method is called, at which time the modules can add toolbar buttons of their own. Then the DiscoveryThread is started, which will periodically look for new event channels and objects in the Trading Service.

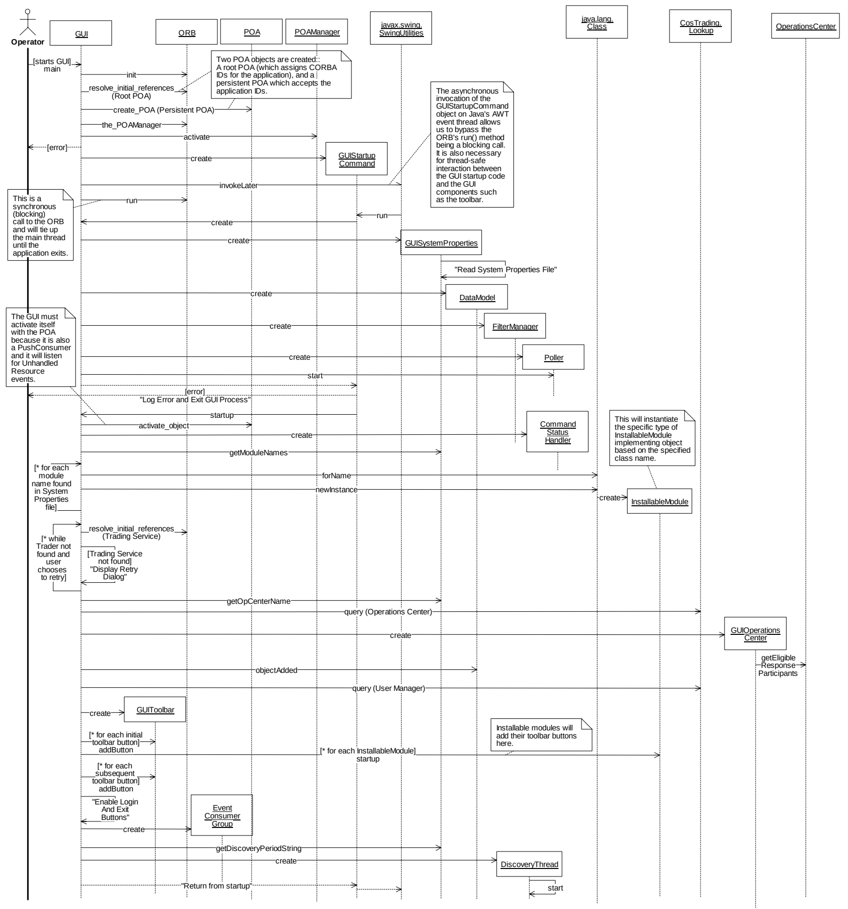

Operator
GUI
ORB
POA
POAManager
javax.swing.SwingUtilities
java.lang.Class
CosTrading.Lookup
OperationsCenter
[starts GUI] main
init
resolve_initial_references (Root POA)
Two POA objects are created:: A root POA (which assigns CORBA IDs for the application), and a persistent POA which accepts the application IDs.
create_POA (Persistent POA)
the_POAManager
activate
[error]
create
GUIStartup Command
invokeLater
The asynchronous invocation of the GUIStartupCommand object on Java's AWT event thread allows us to bypass the ORB's run() method being a blocking call. It is also necessary for thread-safe interaction between the GUI startup code and the GUI components such as the toolbar.
This is a synchronous (blocking) call to the ORB and will tie up the main thread until the application exits.
run
run
create
create
GUISystemProperties
"Read System Properties File"
create
DataModel
The GUI must activate itself with the POA because it is also a PushConsumer and it will listen for Unhandled Resource events.
create
FilterManager
create
Poller
start
[error] "Log Error and Exit GUI Process"
startup
activate_object
create
Command Status Handler
This will instantiate the specific type of InstallableModule implementing object based on the specified class name.
getModuleNames
[* for each module name found in System Properties file]
forName
newInstance
create
InstallableModule
resolve_initial_references (Trading Service)
[* while Trader not found and user chooses to retry]
[Trading Service not found] "Display Retry Dialog"
getOpCenterName
query (Operations Center)
create
GUIOperations Center
getEligible Response Participants
objectAdded
query (User Manager)
create
GUIToolbar
[* for each initial toolbar button] addButton
Installable modules will add their toolbar buttons here.
[* for each InstallableModule] startup
[* for each subsequent toolbar button] addButton
"Enable Login And Exit Buttons"
create
Event Consumer Group
getDiscoveryPeriodString
create
DiscoveryThread
"Return from startup"
start

Figure 23. GUI:StartupBasic (Sequence Diagram)

3.5.2.12 GUI:SystemCommandBasic (Sequence Diagram)

This diagram shows how a system command is handled. A system command is one which does not apply to any served CORBA objects. (For those commands, see the GUI:CommandObjectBasic diagram). First, a context menu is invoked by the user when there are no objects selected (see the GUI:MakeMenuNoneSelected for details on how the menu is made). The GUI, or an InstallableModule, will be attached to the menu items as an ActionListener when the menu is built. When the user clicks on the menu item, Java will invoke the actionPerformed() method of the ActionListener implementing class, which will allow the ActionListener to execute the command.

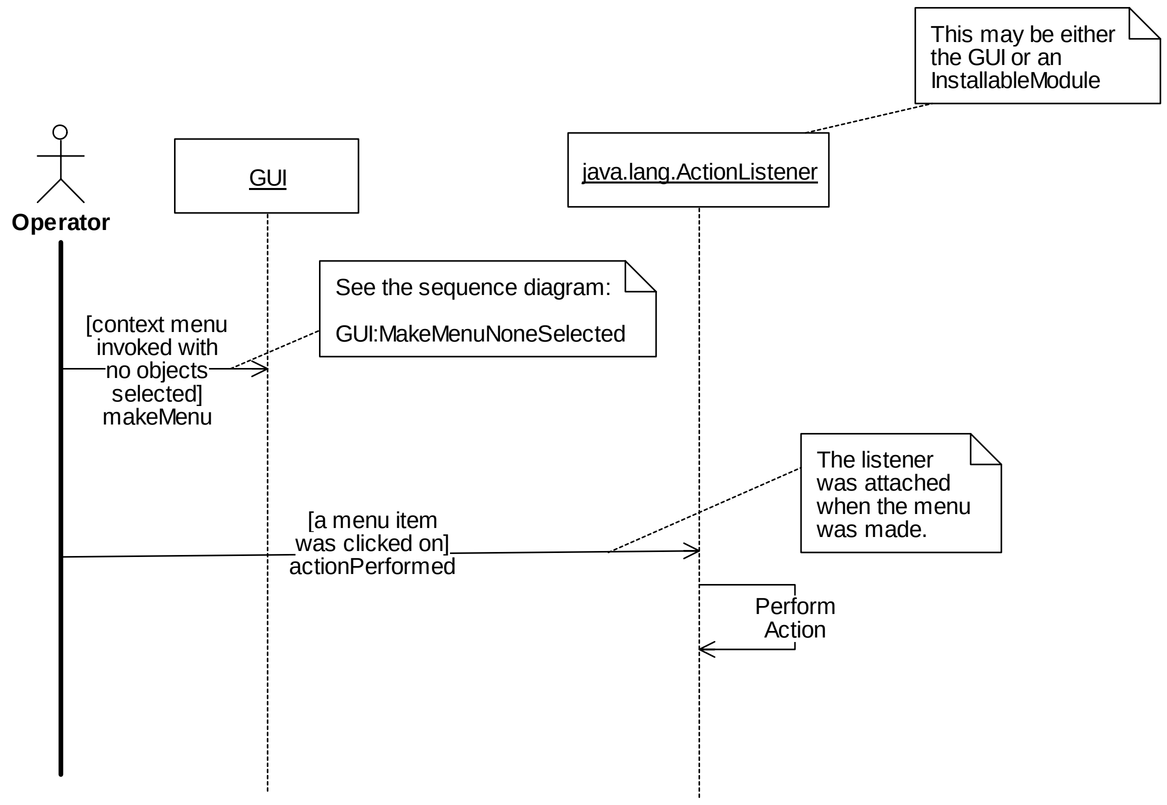

This may be either the GUI or an InstallableModule
Operator
GUI
java.lang.ActionListener
See the sequence diagram:
GUI:MakeMenuNoneSelected
[context menu invoked with no objects selected] makeMenu
The listener was attached when the menu was made.
[a menu item was clicked on] actionPerformed
Perform Action

Figure 24. GUI:SystemCommandBasic (Sequence Diagram)

3.6 GUIDMSModule

3.6.1 Class Diagrams

3.6.1.1 DMSDialogs (Class Diagram)

This diagram shows all of the classes representing GUI windows that exist within the GUIDMSModule.

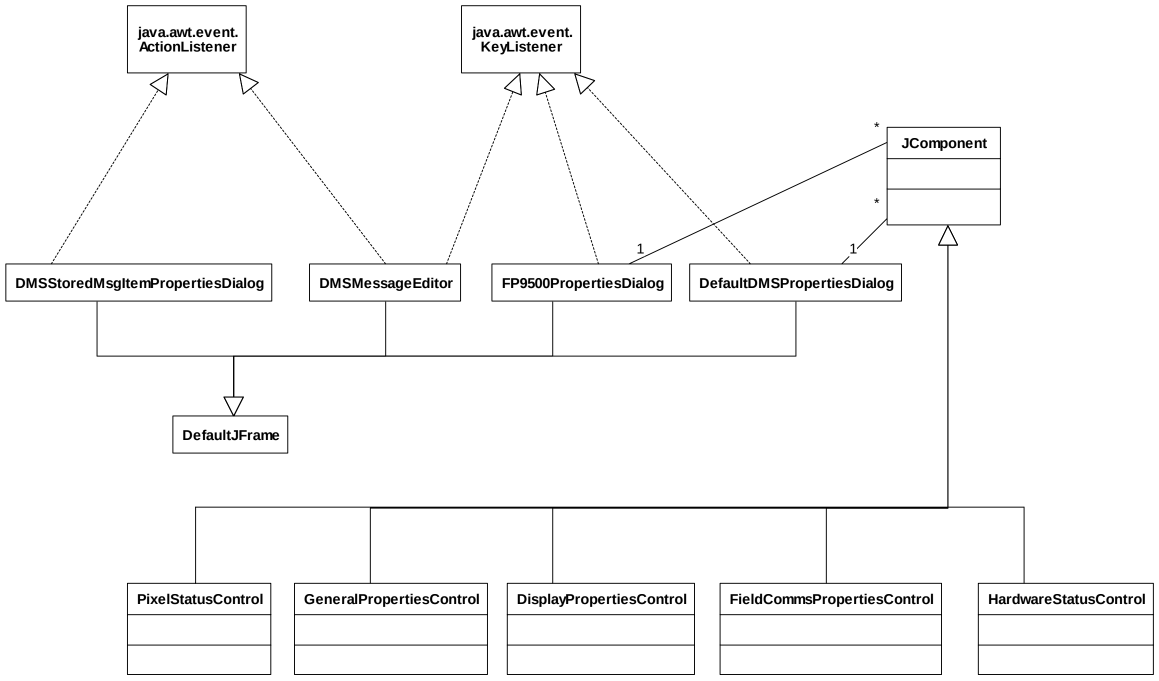


Figure 25. DMSDialogs (Class Diagram)

3.6.1.1.1 DefaultDMSPropertiesDialog (Class)

This dialog is used to view and edit the DMS properties of those models that support a standard set of DMS operational parameters and status information. It uses the control classes derived from JComponent for formatting, display and user editing features.

3.6.1.1.2 DefaultJFrame (Class)

This class provides a default implementation of the WindowManageable interface, and may be used as a base class for other frame windows in the GUI. It handles all interactions with the WindowManager for attaching and detaching, as well as saving the window position.

3.6.1.1.3 DMSMessageEditor (Class)

This class is responsible for allowing an operator to set the current message on a DMS. It also updates a MessageView to allow the operator to preview the message, as it will look on the selected sign, prior to sending the message to the sign controller.

3.6.1.1.4 DMSStoredMsgItemPropertiesDialog (Class)

This dialog is used for creation, viewing and editing of the properties of DMSStoredMsgItem and GUIDMSStoredMsgItem objects.

3.6.1.1.5 FP9500PropertiesDialog (Class)

This dialog is used to view and edit the FP9500 DMS configuration information. It also allows the FP9500 DMS extended status information to be viewed. It delegates the formatting, display and user editing functions to the classes derived from JComponent like GeneralPropertiesControl, PixelStatusControl and other control classes. The control classes used by this class depend on the configuration and status information supported by the FP9500 DMS model.

3.6.1.1.6 DisplayPropertiesControl (Class)

This class is derived from JComponent and is capable of graphical display of DMS display Properties and allows the user to edit these properties. Some examples of DMS display properties are sign height, sign width, character height and character width.

3.6.1.1.7 FieldCommsPropertiesControl (Class)

This class is derived from JComponent and is capable of graphical display of DMS field communication properties and allows the user to edit these properties. Some examples of DMS field communication properties are DMS phone number and comm loss time.

3.6.1.1.8 GeneralPropertiesControl (Class)

This class is derived from JComponent and is capable of graphical display of general DMS Properties and allows the user to edit these properties. Some examples of general DMS properties are DMS name, DMS type and DMS location.

3.6.1.1.9 HardwareStatusControl (Class)

This class is derived from JComponent and is capable of graphical display of DMS controller status. It does not allow the user to edit the information displayed.

3.6.1.1.10 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.6.1.1.11 java.awt.event. KeyListener (Class)

Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

3.6.1.1.12 PixelStatusControl (Class)

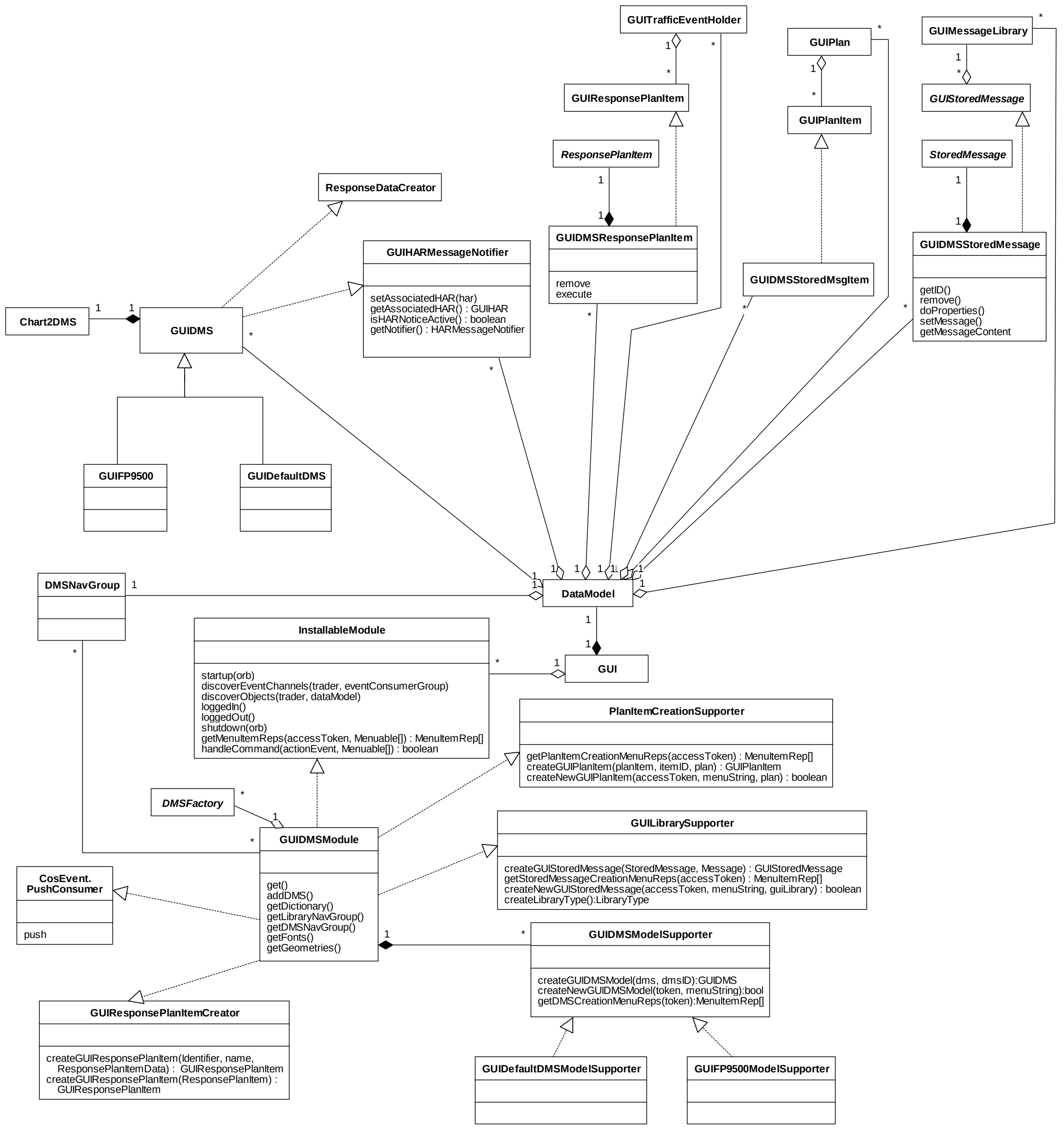
This class is derived from JComponent and is capable of graphical display of DMS Pixel status. It does not allow the user to edit the information displayed.

3.6.1.1.13 JComponent (Class)

This is a Java Swing base class that may be derived by any class having a graphical representation that can be displayed on the screen and that can interact with the user.

3.6.1.2 DMSModuleArchitecture (Class Diagram)

This diagram shows the data hierarchy of the GUIDMSModule and the objects that it supports. The GUIDMSModule:NavigatorSupport class diagram shows how these objects are laid out on the GUI navigator.



GUITrafficEventHolder
GUIPlan
GUIMessageLibrary
GUIResponsePlanItem
GUIPlanItem
GUIStoredMessage
ResponsePlanItem
StoredMessage
ResponseDataCreator
GUIDMSResponsePlanItem
remove
execute
GUIHARMessageNotifier
setAssociatedHAR(har)
getAssociatedHAR() : GUIHAR
isHARNoticeActive() : boolean
getNotifier() : HARMessageNotifier
GUIDMSStoredMsgItem
GUIDMSStoredMessage
getID()
remove()
doProperties()
setMessage()
getMessageContent
Chart2DMS
GUIDMS
GUIFP9500
GUIDefaultDMS
DMSNavGroup
DataModel
InstallableModule
startup(orb)
discoverEventChannels(trader, eventConsumerGroup)
discoverObjects(trader, dataModel)
loggedIn()
loggedOut()
shutdown(orb)
getMenuItemReps(accessToken, Menuable[]) : MenuItemRep[]
handleCommand(actionEvent, Menuable[]) : boolean
GUI
PlanItemCreationSupporter
getPlanItemCreationMenuReps(accessToken) : MenuItemRep[]
createGUIPlanItem(planItem, itemID, plan) : GUIPlanItem
createNewGUIPlanItem(accessToken, menuString, plan) : boolean
DMSFactory
GUIDMSModule
get()
addDMS()
getDictionary()
getLibraryNavGroup()
getDMSNavGroup()
getFonts()
getGeometries()
GUILibrarySupporter
createGUIStoredMessage(StoredMessage, Message) : GUIStoredMessage
getStoredMessageCreationMenuReps(accessToken) : MenuItemRep[]
createNewGUIStoredMessage(accessToken, menuString, guiLibrary) : boolean
createLibraryType():LibraryType
CosEvent.
PushConsumer
push
GUIDMSModelSupporter
createGUIDMSModel(dms, dmsID):GUIDMS
createNewGUIDMSModel(token, menuString):bool
getDMSCreationMenuReps(token):MenuItemRep[]
GUIResponsePlanItemCreator
createGUIResponsePlanItem(Identifier, name,
ResponsePlanItemData) : GUIResponsePlanItem
createGUIResponsePlanItem(ResponsePlanItem) :
GUIResponsePlanItem
GUIDefaultDMSModelSupporter
GUIFP9500ModelSupporter
1
*

Figure 26. DMSModuleArchitecture (Class Diagram)

3.6.1.2.1 Chart2DMS (Class)

The Chart2DMS class extends the DMS interface and defines a more detailed interface to be used in manipulating the Chart II-specific DMS objects within Chart II. It provides a method for getting the DMSArbitrationQueue for a Chart II DMS, which can then be used by traffic events to provide input as to what each traffic event desires to be on the sign. It also provides a method to perform testing on a sign. This method can be extended by derived classes for specific models of signs, which know how to perform certain types of testing on their specific model of sign. Chart II business rules include concepts such as shared resources, arbitration queues, and linking devices usage to traffic events, concepts which go beyond what would be industry-standard DMS control.

3.6.1.2.2 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.6.1.2.3 DMSFactory (Class)

The DMSFactory class specifies the interface to be used to create DMS objects within the Chart II system. It also provides a method to get a list of DMS devices currently in the system.

3.6.1.2.4 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.6.1.2.5 GUIDMSModule (Class)

The GUIDMSModule is an installable module in the GUI that handles all of the DMS specific functionality. Only one GUIDMSModule object may exist within the GUI. This class implements the interfaces to support the frameworks of the GUIPlanModule, the GUILibraryModule, and the GUITrafficEventModule. It handles the creation of model specific GUI DMS objects using the model supporters.

3.6.1.2.6 GUIHARMessageNotifier (Class)

This interface is similar to the HARMessageNotifier interface in that it is implemented by all of the message notifier classes, but this interface is specific to the GUIHARModule and its usage of the GUI wrapper objects.

3.6.1.2.7 GUIDMS (Class)

This class is a GUI “wrapper” object that is used to wrap a CHART2DMS object. This is a abstract class that needs to be extended by the GUI DMS model specific classes.

3.6.1.2.8 GUILibrarySupporter (Class)

This class allows the GUILibraryModule to maintain stored messages that have differing formats. When an object of this type is installed the user can create, maintain, and use the specific type of libraries and stored messages that the object supports.

3.6.1.2.9 ResponseDataCreator (Class)

This interface enables the creation of type-specific ResponsePlanItemData objects, which are used for creating the appropriate type of ResponsePlanItem. An object implementing this interface can be added to the response plan of a traffic event. Implementers of this interface include plan items and response devices.

3.6.1.2.10 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.6.1.2.11 DMSNavGroup (Class)

This class serves as a container for all of the GUIDMS objects in the GUIDMSModule, when they are displayed in the Navigator. The GUIDMSModule has one instance of this class.

3.6.1.2.12 GUIDMSResponsePlanItem (Class)

This class is a GUI “wrapper” object that is used to wrap a ResponsePlanItem object which contains a DMSRPIData object.

3.6.1.2.13 GUIDMSStoredMessage (Class)

This class is a GUI “wrapper” object that is used to wrap a StoredMessage object of DMSMessage type. It helps in the creation of a DMS stored message using a DMSMessageEditor.

3.6.1.2.14 GUIPlanItem (Class)

This is a GUI base class for all the plan items. Each GIUPlanItem object will serve as a GUI wrapper to cache the plan item data locally and also to handle all user interaction in the GUI, such as menus and command handling.

3.6.1.2.15 GUIResponsePlanItem (Class)

This is a base class for the GUI wrapper object that is used to wrap a ResponsePlanItem. The ResponsePlanItem represents a proposed action to perform on a target object in response to a TrafficEvent. This wrapper object adds GUI-specific functionality to the response plan item.

3.6.1.2.16 GUIPlan (Class)

This class is a GUI wrapper for the Plan object. The wrapping is done to cache the data locally for faster access, as well as to give the Plan some GUI-specific functionality such as menus and command handling.

3.6.1.2.17 GUIStoredMessage (Class)

This class is a GUI "wrapper" object that is used to wrap a StoredMessage object. It provides a user interface object which can implement whatever interfaces are necessary for the object to exist within the GUI framework (for example, an object must support the NavTreeDisplayable and/or NavListDisplayable interface to be displayed in the Navigator).

3.6.1.2.18 GUITrafficEventHolder (Class)

This object represents a TrafficEvent and provides GUI functionality for the TrafficEvent. This class contains generic data and operations that apply to any type of TrafficEvent. It also "holds" a type-specific GUITrafficEvent. If the type of the TrafficEvent is changed, the old GUITrafficEvent object (stored within this "holder" class) will be switched out for a new GUITrafficEvent of a different type, but the GUITrafficEventHolder will remain in existence.

3.6.1.2.19 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description which are used to allow the user to organize messages.

3.6.1.2.20 GUIDefaultDMS (Class)

This class is derived from the GUIDMS class and represents a standard model DMS. This class can handle the configuration requirements and status information that are standard across all DMS types.

3.6.1.2.21 GUIDMSModelSupporter (Class)

This interface must be implemented by any class that intends to provide functionality for the creation of DMS objects of a specific model type. The GUIDMSModelSupporter provides methods to return the specific menu string, which when selected on the GUI by the user, results in the creation of the DMS object of that type. There are methods in the interface that help in the creation of the model specific DMS object.

3.6.1.2.22 GUIDMSStoredMsgItem (Class)

This class is a GUI “wrapper” object that is used to wrap a PlanItem object which contains the DMSPlanItemData. It helps in the creation of a DMS plan item data using the DMSStoredMsgItemProperties object.

3.6.1.2.23 GUIFP9500 (Class)

This class is derived from the GUIDMS class and represents a FP9500 model type DMS. This class can handle the specialized configuration requirements of a FP9500 model DMS and interpret the model specific status information.

3.6.1.2.24 GUIResponsePlanItemCreator (Class)

This interface is used to enable the creation of specific types of GUIResponsePlanItem wrapper objects depending upon which type of ResponsePlanItem is being wrapped. Any class wishing to create GUIResponsePlanItems must implement this interface and add themselves to the GUITrafficEventModule at GUI startup time. When the GUITrafficEventModule discovers a ResponsePlanItem or catches a CORBA event indicating that a new response plan item has been created, it will call each known GUIResponsePlanItemCreator to give it an opportunity to create a specific type of GUI wrapper object.

3.6.1.2.25 PlanItemCreationSupporter (Class)

This interface must be implemented in any modules that wish to support the plan module. The modules must attach their PlanItemCreationSupporters at startup. The GUIPlanModule will then call the supporter when it is time to display the Plan menu or to create a specific type of plan item or GUIPlanItem.

3.6.1.2.26 GUIDefaultDMSModelSupporter (Class)

This class provides functionality for the creation of a standard DMS model object, by implementing the GUIDMSModelSupporter interface.

3.6.1.2.27 GUIFP9500ModelSupporter (Class)

This class provides functionality for the creation of FP9500 type DMS object, by implementing the GUIDMSModelSupporter interface.

3.6.1.2.28 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.6.1.2.29 GUIMessageLibrary (Class)

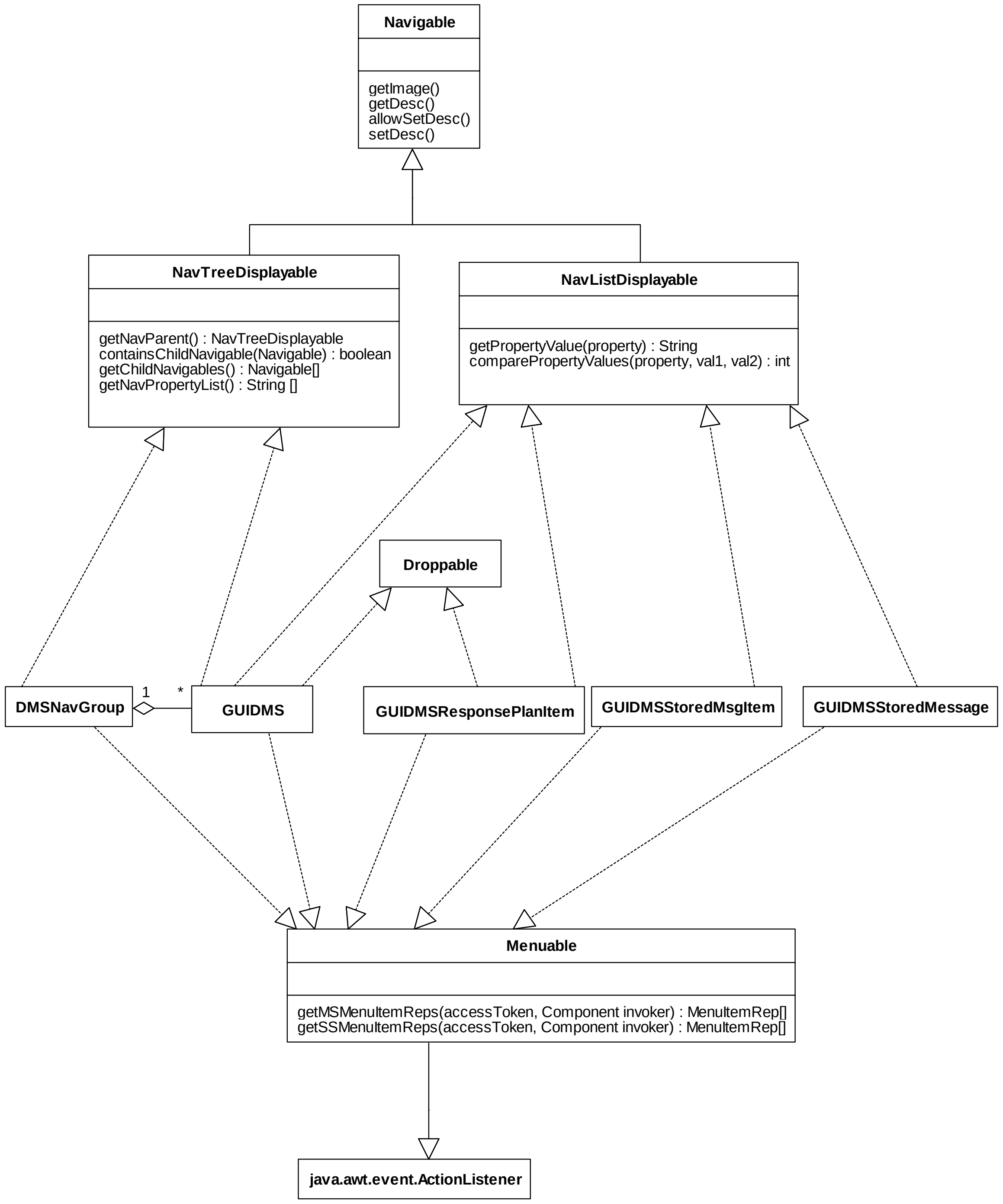
This class is a GUI “wrapper” object that is used to wrap a MessageLibrary object. The wrapping is done to cache the data locally for faster access, as well as to give the MessageLibrary some GUI-specific functionality such as menus and command handling.

3.6.1.2.30 ResponsePlanItem (Class)

Objects of this type can be executed as part of a traffic event response plan. A ResponsePlanItem can be executed by an operator, at which time it becomes the responsibility of the System to activate the item on the ResponseDevice as soon as it is appropriate.

3.6.1.3 DMSNavigatorSupport (Class Diagram)

This diagram shows the user interface relationships of the objects supported by the GUIDMSModule.



Navigable
getImage()
getDesc()
allowSetDesc()
setDesc()
NavTreeDisplayable
getNavParent() : NavTreeDisplayable
containsChildNavigable(Navigable) : boolean
getChildNavigables() : Navigable[]
getNavPropertyList() : String []
NavListDisplayable
getPropertyValue(property) : String
comparePropertyValues(property, val1, val2) : int
Droppable
DMSNavGroup
1
*
GUIDMS
GUIDMSResponsePlanItem
GUIDMSStoredMsgItem
GUIDMSStoredMessage
Menuable
getMSMenuItemReps(accessToken, Component invoker) : MenuItemRep[]
getSSMenuItemReps(accessToken, Component invoker) : MenuItemRep[]
java.awt.event.ActionListener

Figure 27. DMSNavigatorSupport (Class Diagram)

3.6.1.3.1 DMSNavGroup (Class)

This class serves as a container for all of the GUIDMS objects in the GUIDMSModule, when they are displayed in the Navigator. The GUIDMSModule has one instance of this class.

3.6.1.3.2 GUIDMS (Class)

This class is a GUI “wrapper” object that is used to wrap a CHART2DMS object. This is a abstract class that needs to be extended by the GUI DMS model specific classes.

3.6.1.3.3 GUIDMSResponsePlanItem (Class)

This class is a GUI “wrapper” object that is used to wrap a ResponsePlanItem object which contains a DMSRPIData object.

3.6.1.3.4 GUIDMSStoredMsgItem (Class)

This class is a GUI “wrapper” object that is used to wrap a PlanItem object which contains the DMSPlanItemData. It helps in the creation of a DMS plan item data using the DMSStoredMsgItemProperties object.

3.6.1.3.5 GUIDMSStoredMessage (Class)

This class is a GUI “wrapper” object that is used to wrap a StoredMessage object of DMSMessage type. It helps in the creation of a DMS stored message using a DMSMessageEditor.

3.6.1.3.6 Droppable (Class)

This interface must be implemented by any object wishing to take part in a drag and drop operation. It is used by the DropHandler class to determine if a drop action should be allowed and to delegate the handling of the drop action after it is performed.

3.6.1.3.7 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.6.1.3.8 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

3.6.1.3.9 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.6.1.3.10 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

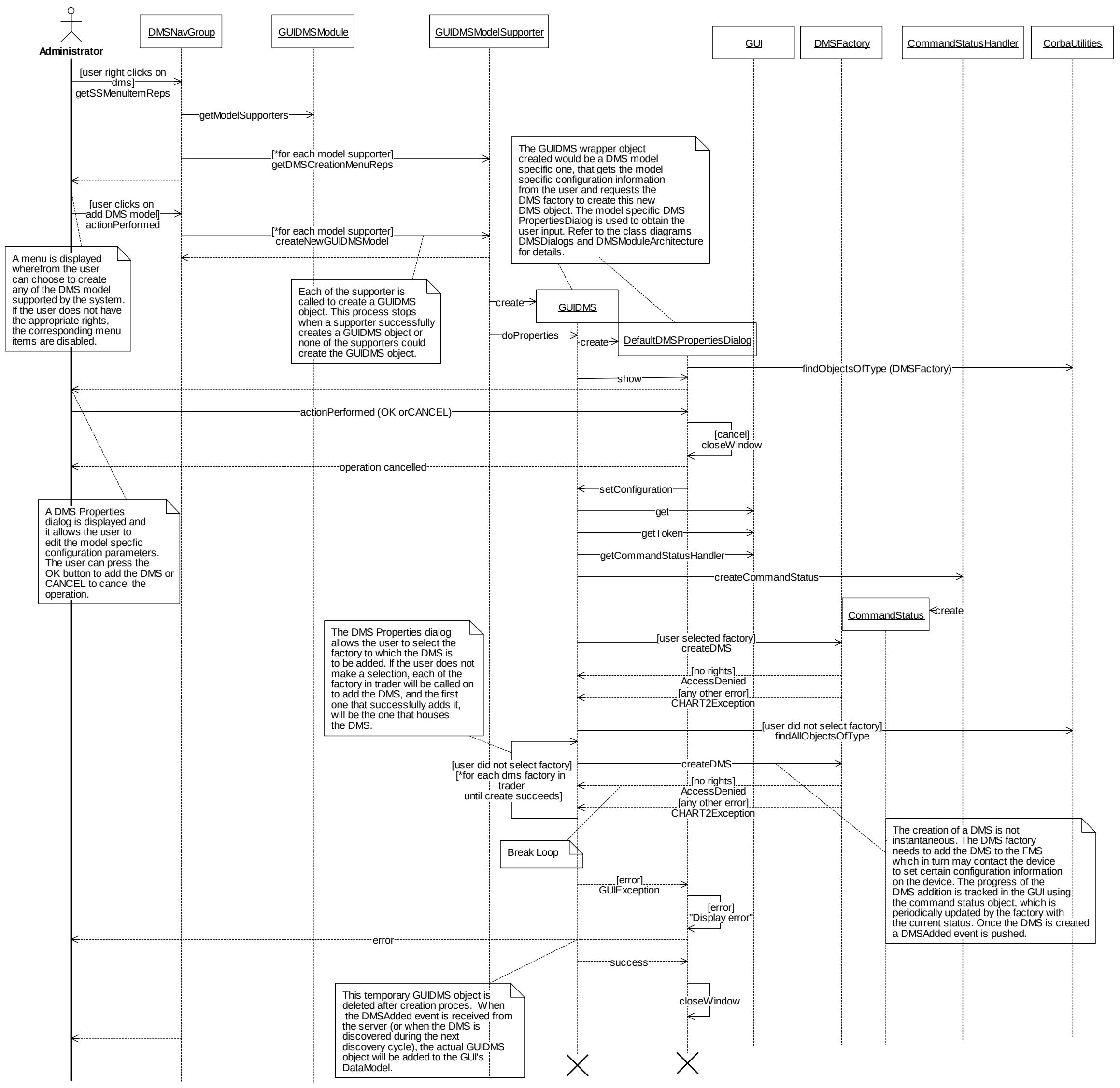
3.6.1.3.11 Navigable (Class)

This interface will be implemented by any class that supports the Navigator on either the left or right side (the tree or list view). This includes the functionality common to both the tree and list.

3.6.2 Sequence Diagrams

3.6.2.1 GUIDMSModule:AddDMS (Sequence Diagram)

This sequence shows how an operator adds a new DMS to the system. The processing shown here is for adding a DMS of the default type. The processing involved in adding a model specific DMS is much the same, except that the GUI DMS and the Properties Dialog objects are the model specific derivatives. When the user right clicks on the GUIDMS object, the model supporters registered in the system are called on to return the menu item string that is to be displayed to the user in order to create the DMS model that it supports. The operator selects the DMS model that he/she wishes to create. If the user does not have the appropriate functional rights, the corresponding menu items are disabled. All the DMS model supporters are then called upon to create a GUIDMS object. Only one of the supporters will identify the menu item that was selected (in this case the default supporter), and proceeds to create the appropriate DMS model. The operator will be shown a DMS properties dialog box with default configuration information which he/she may modify to alter the configuration of the DMS. When the operator presses OK, the new DMS will be added to the system. The DMS will be added to the DMS factory that the user selected in the properties dialog. If the user did not make a selection, each of the factories in the CORBA trader is called one by one (starting with the factory closest to the GUI’s trader) to create the DMS, and the first factory that successfully creates it, will be the home of the DMS. Once the DMS is added successfully, a DMSAdded event will be pushed from the server through the DMS event channel and the new GUIDMS object will be added to the DataModel, which will update all windows after a short delay.



Administrator
DMSNavGroup
GUIDMSModule
GUIDMSModelSupporter
GUI
DMSFactory
CommandStatusHandler
CorbaUtilities
[user right clicks on dms] getSSMenuItemReps
getModelSupporters
[*for each model supporter] getDMSCreationMenuReps
[user clicks on add DMS model] actionPerformed
[*for each model supporter] createNewGUIDMSModel
A menu is displayed wherefrom the user can choose to create any of the DMS model supported by the system. If the user does not have the appropriate rights, the corresponding menu items are disabled.
Each of the supporter is called to create a GUIDMS object. This process stops when a supporter successfully creates a GUIDMS object or none of the supporters could create the GUIDMS object.
The GUIDMS wrapper object created would be a DMS model specific one, that gets the model specific configuration information from the user and requests the DMS factory to create this new DMS object. The model specific DMS PropertiesDialog is used to obtain the user input. Refer to the class diagrams DMSDialogs and DMSModuleArchitecture for details.
create
GUIDMS
doProperties
create
DefaultDMSPropertiesDialog
findObjectsOfType (DMSFactory)
show
actionPerformed (OK orCANCEL)
[cancel] closeWindow
operation cancelled
setConfiguration
get
getToken
getCommandStatusHandler
createCommandStatus
A DMS Properties dialog is displayed and it allows the user to edit the model specfic configuration parameters. The user can press the OK button to add the DMS or CANCEL to cancel the operation.
CommandStatus
create
The DMS Properties dialog allows the user to select the factory to which the DMS is to be added. If the user does not make a selection, each of the factory in trader will be called on to add the DMS, and the first one that successfully adds it, will be the one that houses the DMS.
[user selected factory] createDMS
[no rights] AccessDenied
[any other error] CHART2Exception
[user did not select factory] findAllObjectsOfType
[user did not select factory] [*for each dms factory in trader until create succeeds]
createDMS
[no rights] AccessDenied
[any other error] CHART2Exception
The creation of a DMS is not instantaneous. The DMS factory needs to add the DMS to the FMS which in turn may contact the device to set certain configuration information on the device. The progress of the DMS addition is tracked in the GUI using the command status object, which is periodically updated by the factory with the current status. Once the DMS is created a DMSAdded event is pushed.
Break Loop
[error] GUIException
[error] "Display error"
error
success
closeWindow
This temporary GUIDMS object is deleted after creation proces. When the DMSAdded event is received from the server (or when the DMS is discovered during the next discovery cycle), the actual GUIDMS object will be added to the GUI's DataModel.

Figure 28. GUIDMSModule:AddDMS (Sequence Diagram)

3.6.2.2 GUIDMSModule:AddDMSStoredMessageItem (Sequence Diagram)

This diagram shows how a PlanItem is added to the system. The user clicks on the GUIPlan object in the Navigator and chooses “Create DMS Plan Item”. The GUIPlan then calls all the PlanItemCreationSupporters (the GUIDMSModule is one) to create the GUIPlanItem. The processing stops with the first supporter that returns the new object after successful creation. The menu selection being a DMS plan item, the GUIDMSModule recognizes the menu item string. The module creates a temporary GUI wrapper for a plan item and calls it to display its properties, which invokes the DMSStoredMsgItemPropertiesDialog. When the user clicks Apply or OK, the dialog calls back to the GUIDMSStoredMsgItem wrapper object to set the item data. Since the wrapper contains no served PlanItem, it calls the CHART2DMS to create one. If successful, the server will push a PlanItemAdded event to all GUIs, which the GUI will catch to create a new GUIDMSStoredMsgItem wrapper object (the temporary wrapper will be deleted).

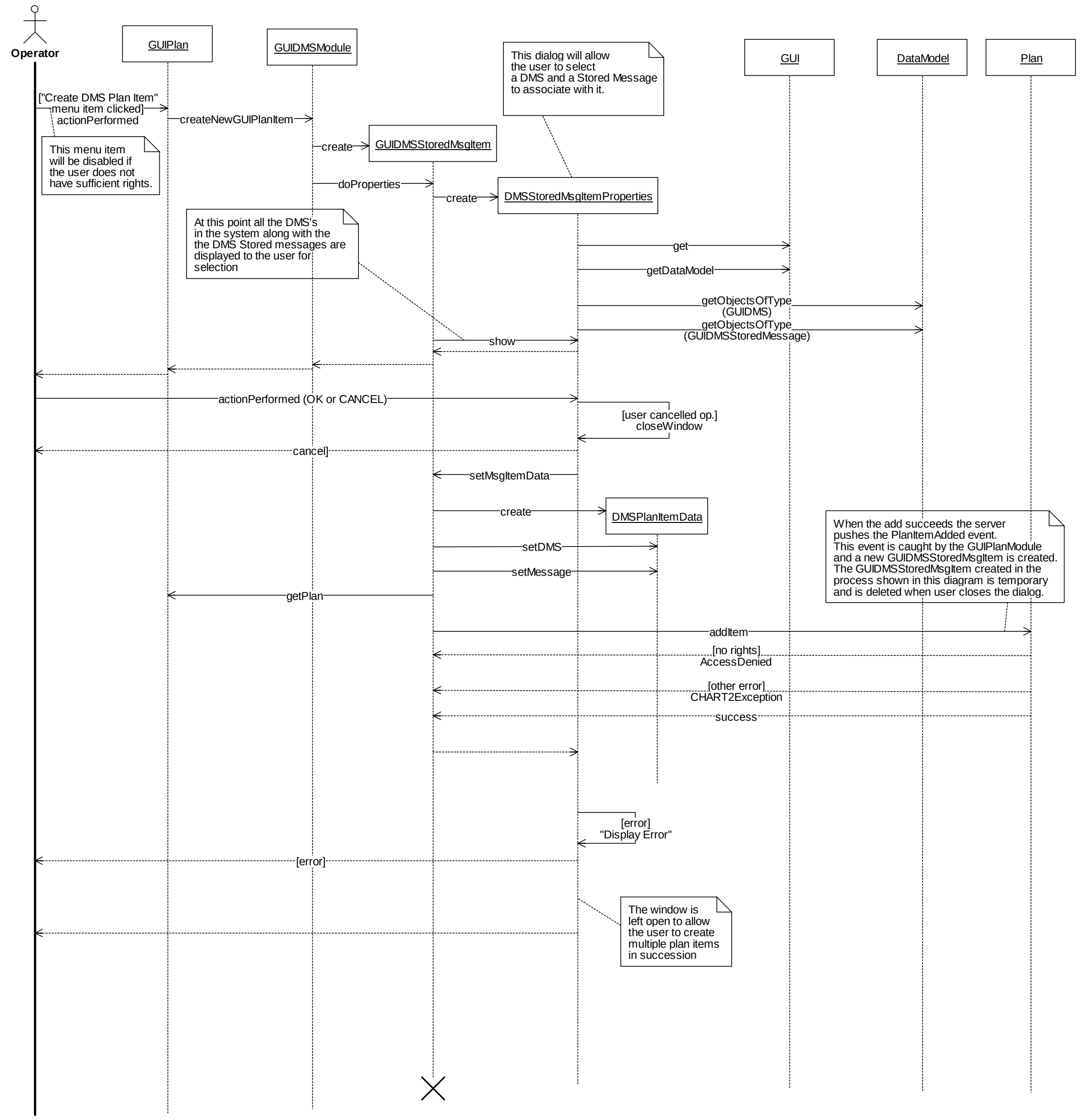

Operator
GUIPlan
GUIDMSModule
GUI
DataModel
Plan
This dialog will allow the user to select a DMS and a Stored Message to associate with it.
["Create DMS Plan Item" menu item clicked] actionPerformed
createNewGUIPlanItem
create
GUIDMSStoredMsgItem
This menu item will be disabled if the user does not have sufficient rights.
doProperties
create
DMSStoredMsgItemProperties
At this point all the DMS's in the system along with the the DMS Stored messages are displayed to the user for selection
get
getDataModel
getObjectsOfType (GUIDMS)
getObjectsOfType (GUIDMSStoredMessage)
show
actionPerformed (OK or CANCEL)
[user cancelled op.] closeWindow
cancel]
setMsgItemData
create
DMSPlanItemData
When the add succeeds the server pushes the PlanItemAdded event. This event is caught by the GUIPlanModule and a new GUIDMSStoredMsgItem is created. The GUIDMSStoredMsgItem created in the process shown in this diagram is temporary and is deleted when user closes the dialog.
setDMS
setMessage
getPlan
addItem
[no rights] AccessDenied
[other error] CHART2Exception
success
[error] "Display Error"
[error]
The window is left open to allow the user to create multiple plan items in succession

Figure 29. GUIDMSModule:AddDMSStoredMessageItem (Sequence Diagram)

3.6.2.3 GUIDMSModule:BlankDMSInMaintenanceMode (Sequence Diagram)

This sequence diagram shows how a user with appropriate rights can blank a DMS when the device is in maintenance mode. The sequence is initiated when the user right clicks on a GUIDMS object in the navigator and selects the “Blank” menu item. The GUIDMS object creates a CommandStatus object and then calls the Chart2DMS object that it wraps to perform the blank operation. The progress of the blank operation is displayed to the user on the command status window, which is updated as and when the server updates the CommandStatus object that was passed to it along with the blank command. If successful, the server will push CORBA events indicating the changed display status.

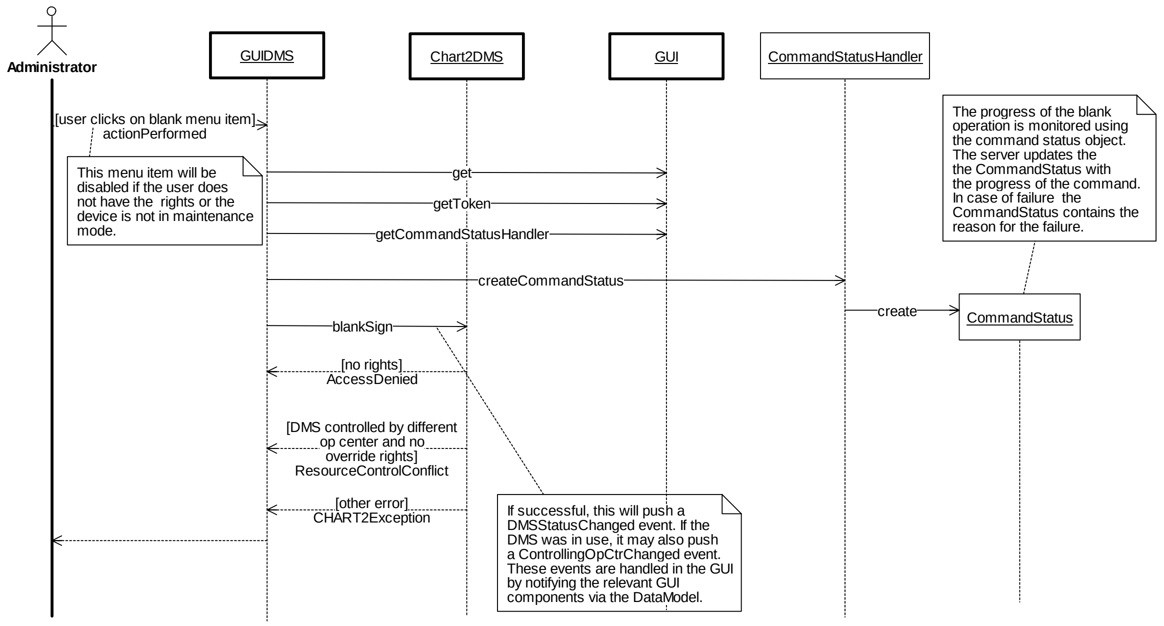

Administrator
GUIDMS
Chart2DMS
GUI
CommandStatusHandler
[user clicks on blank menu item] actionPerformed
This menu item will be disabled if the user does not have the rights or the device is not in maintenance mode.
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
The progress of the blank operation is monitored using the command status object. The server updates the the CommandStatus with the progress of the command. In case of failure the CommandStatus contains the reason for the failure.
blankSign
[no rights] AccessDenied
[DMS controlled by different op center and no override rights] ResourceControlConflict
[other error] CHART2Exception
If successful, this will push a DMSStatusChanged event. If the DMS was in use, it may also push a ControllingOpCtrChanged event. These events are handled in the GUI by notifying the relevant GUI components via the DataModel.

Figure 30. GUIDMSModule:BlankDMSInMaintenanceMode (Sequence Diagram)

3.6.2.4 GUIDMSModule:CreateDMSStoredMessage (Sequence Diagram)

This diagram shows how a DMS stored message is created. When the user right clicks on the GUIMessageLibrary object, the library supporters registered in the system are called on to return the menu item string that is to be displayed to the user in order to create the library message that it supports. In this case, the operator selects the menu item to create a DMS message that is supported by the GUIDMSModule. If the user does not have the appropriate functional rights, the corresponding menu items are disabled. All the library supporters are then called upon to create the library message object. Only one of the supporters will identify the menu item that was selected (in this case it is the GUIDMSModule), and proceeds to create the appropriate library message. The GUIDMSModule creates a temporary GUIDMSStoredMessage object to edit, and calls doProperties to show the DMSMessageEditor dialog. As the user types, banned words will be shown to the user. When the user clicks OK, the message editor will check for disapproved words and provides suggestions to replace these words. A new DMSMessage object is created and the setMessage method is called on the GUIDMSStoredMessage wrapper object. Since the wrapper does not contain a served StoredMessage object, it calls the message library to create one. If successful, the server will create a new StoredMessage object and will push an event to update all of the GUIs.

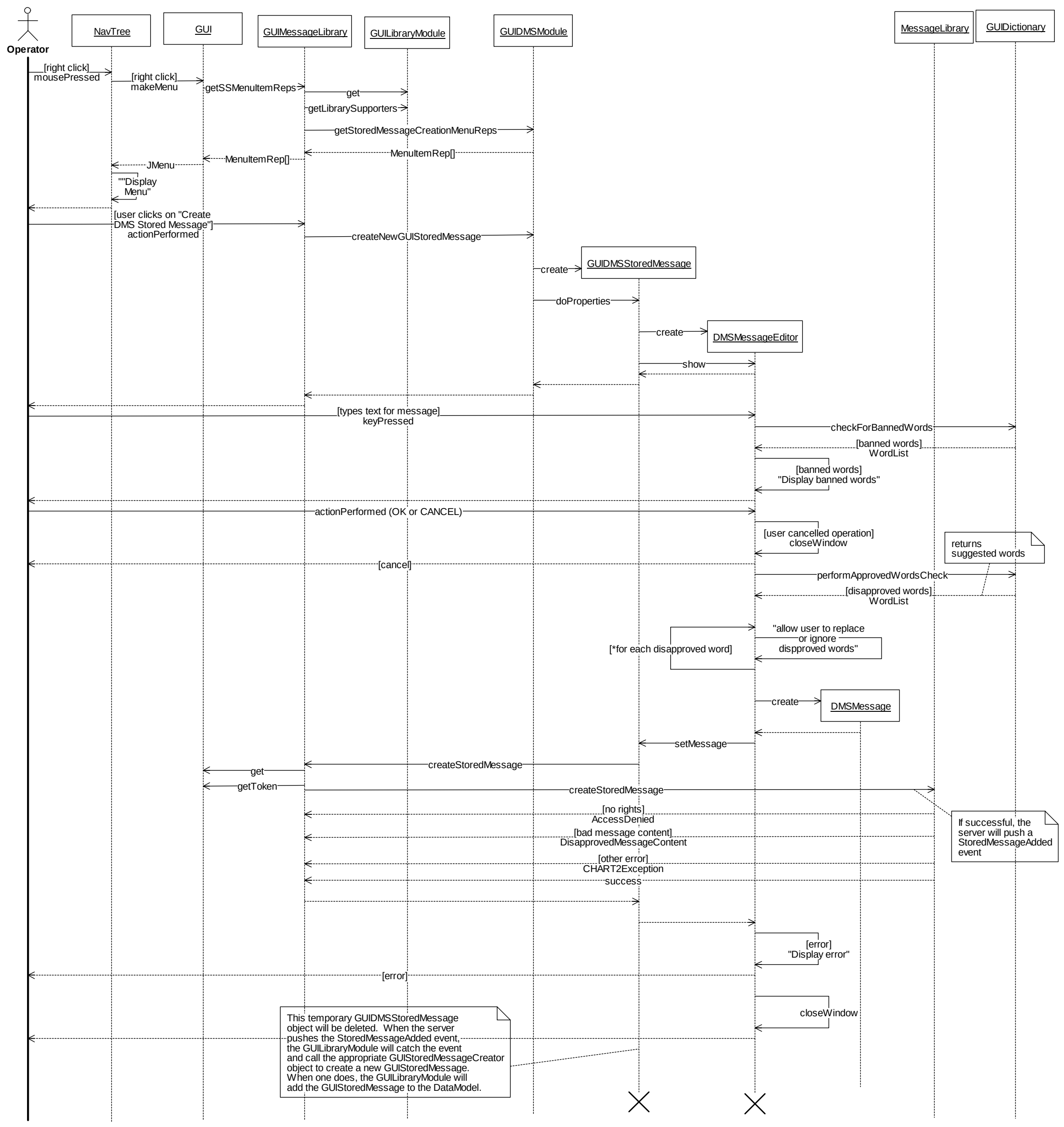
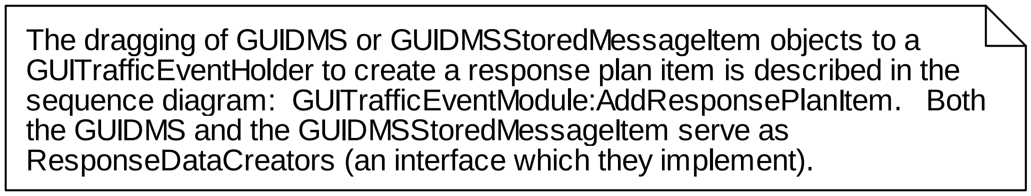


Figure 31. GUIDMSModule:CreateDMSStoredMessage (Sequence Diagram)

3.6.2.5 GUIDMSModule:CreateResponsePlanItem (Sequence Diagram)

This diagram shows how a DMS response plan item is added to the system. The user drags a GUIDMS or a GUIDMSStoredMsgItem object over the GUITrafficEventHolder (the object representing the traffic event in the GUI) and drops it. Since the GUIDMS and GUIDMSStoredMsgItem objects both implement the ResponseDataCreator interface, the GUITrafficEventModule can use either of these to create a DMSResponsePlanItemData, which it then uses to create a ResponsePlanItem. See the sequence diagram: GUITrafficEventModule:AddResponsePlanItem for details.



The dragging of GUIDMS or GUIDMSStoredMessageItem objects to a GUITrafficEventHolder to create a response plan item is described in the sequence diagram: GUITrafficEventModule:AddResponsePlanItem. Both the GUIDMS and the GUIDMSStoredMessageItem serve as ResponseDataCreators (an interface which they implement).

Figure 22. GUIDMSModule:CreateResponsePlanItem (Sequence Diagram)

3.6.2.6 GUIDMSModule:DiscoverEventchannels (Sequence Diagram)

This diagram shows the processing involved in the DMS event channel discovery, which takes place at startup of GUIDMSModule and periodically from thereon. The GUIDMSModule queries the event channels from the trading service, creates a PushConsumer to receive the CORBA events, and adds the PushConsumer objects to the EventConsumerGroup for maintenance of the event channels.

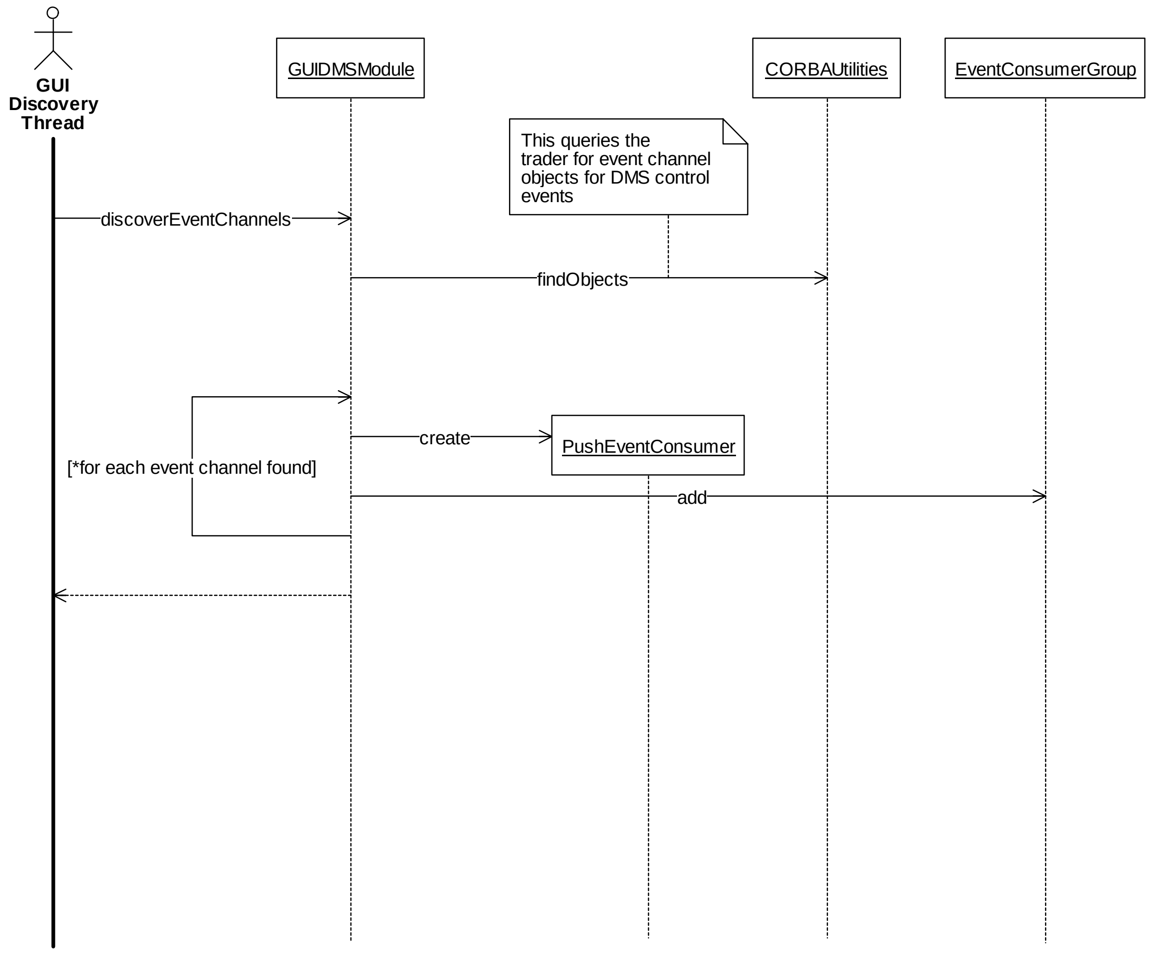


Figure 32. GUIDMSModule:DiscoverEventchannels (Sequence Diagram)

3.6.2.7 GUIDMSModule:DiscoverObjects (Sequence Diagram)

This diagram shows the processing involved in the discovery of Chart2DMS corba objects, which takes place at startup of GUIDMSModule and periodically from thereon. The GUIDMSModule queries the trading service for all the DMS Factory objects. Each factory object is called on to return the DMS objects that it serves. If the object discovered is already in the data model, no action is required. Otherwise, each of the objects discovered in this fashion, is passed on to the GUIDMSModelFactory that is capable of identifying the DMS model type using its collection of GUIDMSModelSupporter objects, and creating a GUIDMS object that wraps this Chart2DMS corba object. Thus the GUIDMS object created in this manner, is model specific (refer to the DMSModuleArchitecture class diagram for the classes that derives from GUIDMS to represent specific DMS models). It is then added to the data model.

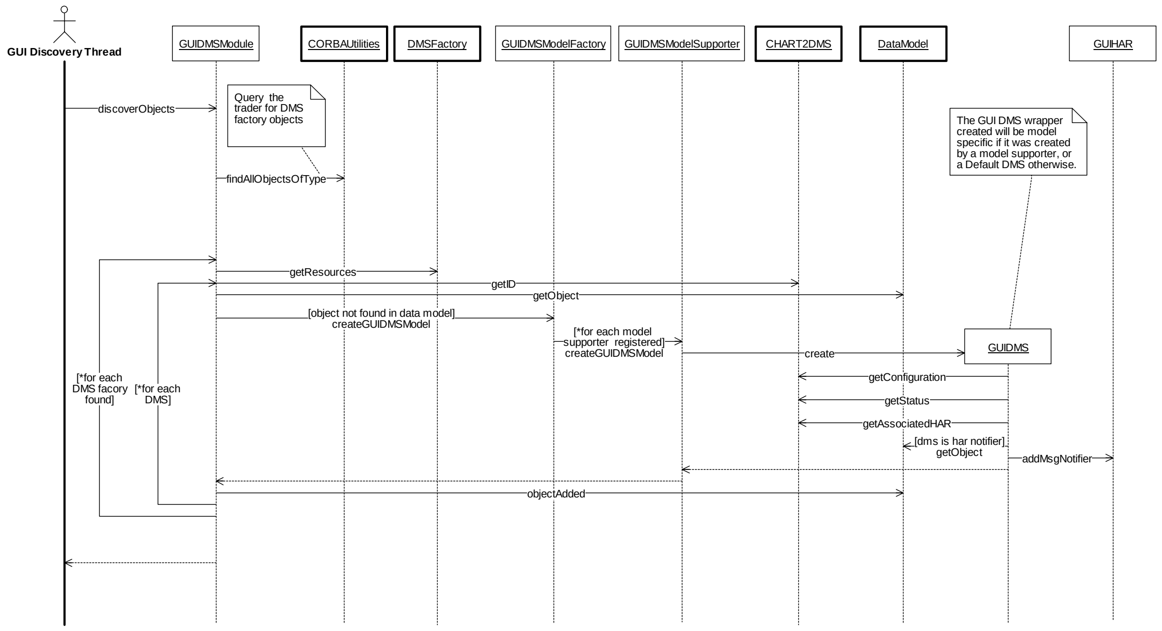

GUI Discovery Thread
GUIDMSModule
CORBAUtilities
DMSFactory
GUIDMSModelFactory
GUIDMSModelSupporter
CHART2DMS
DataModel
GUIHAR
discoverObjects
Query the trader for DMS factory objects
findAllObjectsOfType
The GUI DMS wrapper created will be model specific if it was created by a model supporter, or a Default DMS otherwise.
getResources
getID
getObject
[object not found in data model] createGUIDMSModel
[*for each model supporter registered] createGUIDMSModel
create
GUIDMS
[*for each DMS facory found]
[*for each DMS]
getConfiguration
getStatus
getAssociatedHAR
[dms is har notifier] getObject
addMsgNotifier
objectAdded

Figure 34. GUIDMSModule:DiscoverObjects (Sequence Diagram)

3.6.2.8 GUIDMSModule:DMSRemovedEvent (Sequence Diagram)

This diagram shows how a DMSDeleted event is handled in the GUI. First, an attempt is made to get the GUIDMS object from the DataModel. If it exists, the GUIDMS is removed from the DataModel. The DataModel notifies all the observers about the removal of the DMS from the system. This change will be reflected on all the observer windows.

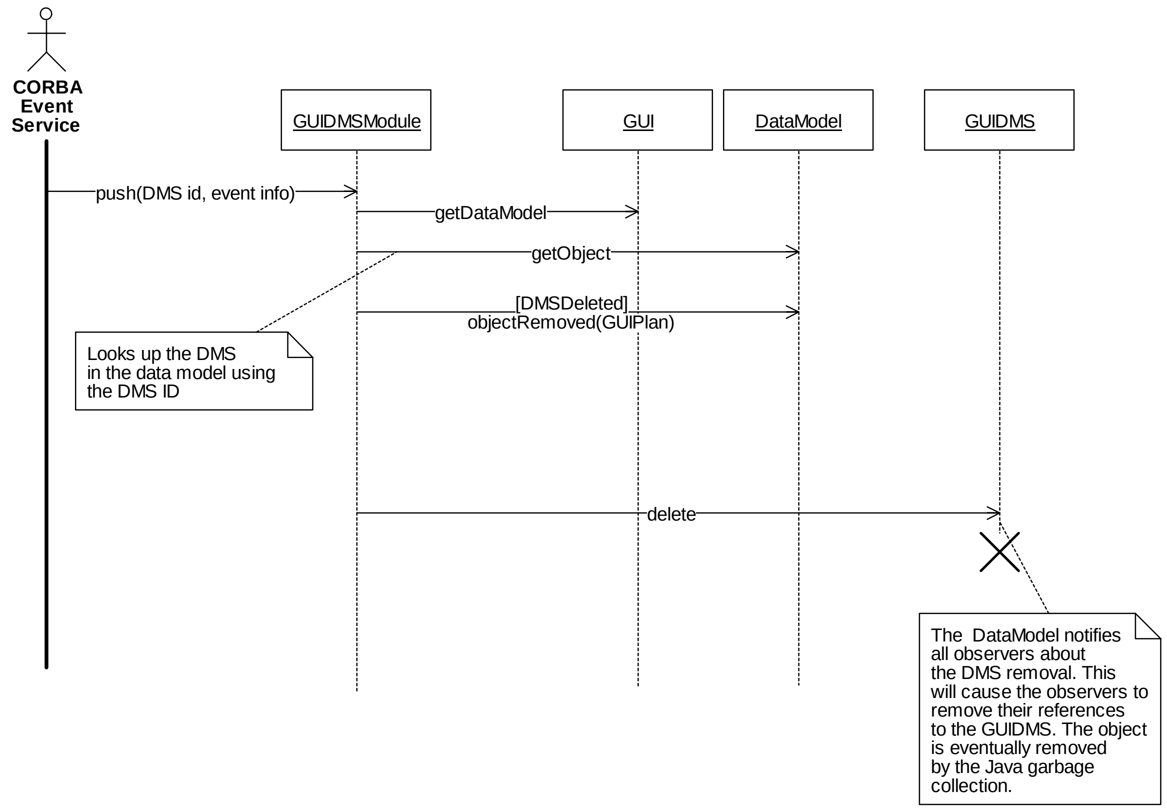

CORBA Event Service
GUIDMSModule
GUI
DataModel
GUIDMS
push(DMS id, event info)
getDataModel
getObject
[DMSDeleted] objectRemoved(GUIPlan)
Looks up the DMS in the data model using the DMS ID
delete
The DataModel notifies all observers about the DMS removal. This will cause the observers to remove their references to the GUIDMS. The object is eventually removed by the Java garbage collection.

Figure 35. GUIDMSModule:DMSRemovedEvent (Sequence Diagram)

3.6.2.9 GUIDMSModule:DMSStateChangeEvents (Sequence Diagram)

This sequence diagram shows the processing involved in handling a DMS configuration change or a DMS Status Change event that is pushed by the server. The event info accompanying the event includes the DMS ID whose state has changed and the changed state information. The GUIDMS object corresponding to the DMS ID in the event info is looked up in the data model. This GUIDMS object updates itself with the current state and alerts the data model, which informs all the registered observers about the change. The data passed on to the GUIDMS is DMS model dependent. Since the GUIDMS is of the same model type as the data, it can interpret the event data and update its state.

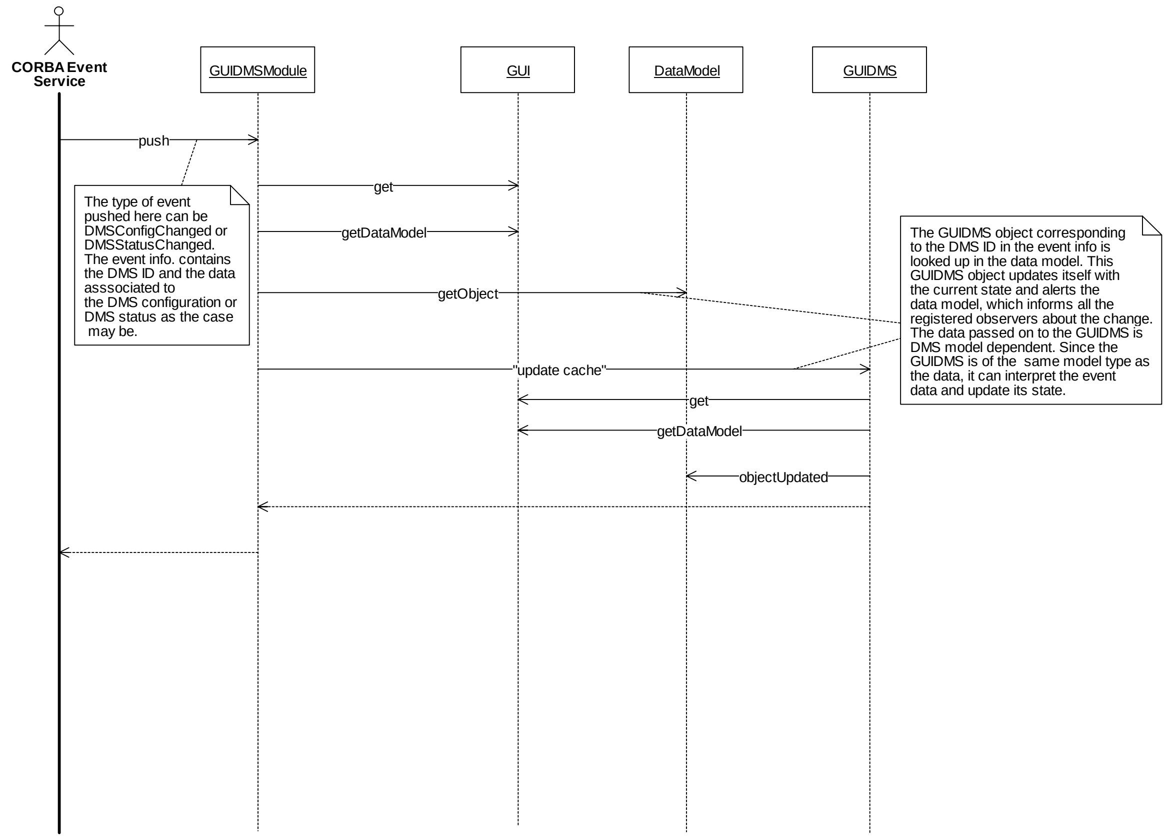

CORBA Event Service
GUIDMSModule
GUI
DataModel
GUIDMS
push
get
getDataModel
getObject
"update cache"
get
getDataModel
objectUpdated
The type of event pushed here can be DMSConfigChanged or DMSStatusChanged. The event info. contains the DMS ID and the data asssociated to the DMS configuration or DMS status as the case may be.
The GUIDMS object corresponding to the DMS ID in the event info is looked up in the data model. This GUIDMS object updates itself with the current state and alerts the data model, which informs all the registered observers about the change. The data passed on to the GUIDMS is DMS model dependent. Since the GUIDMS is of the same model type as the data, it can interpret the event data and update its state.

Figure 36. GUIDMSModule:DMSStateChangeEvents (Sequence Diagram)

3.6.2.10 GUIDMSModule:Login (Sequence Diagram)

This sequence of events is initiated when a user logs in to the system using either the login or change user commands from the toolbar window. These commands cause the GUI:LoginBasic sequence or GUI:ChangeUserBasic to be performed. As part of either of these sequences, the GUI will call each of the installed modules giving them a chance to perform necessary operations to set up data specific to a particular user. The GUIDMSModule does not currently need to perform any processing when a user logs in.

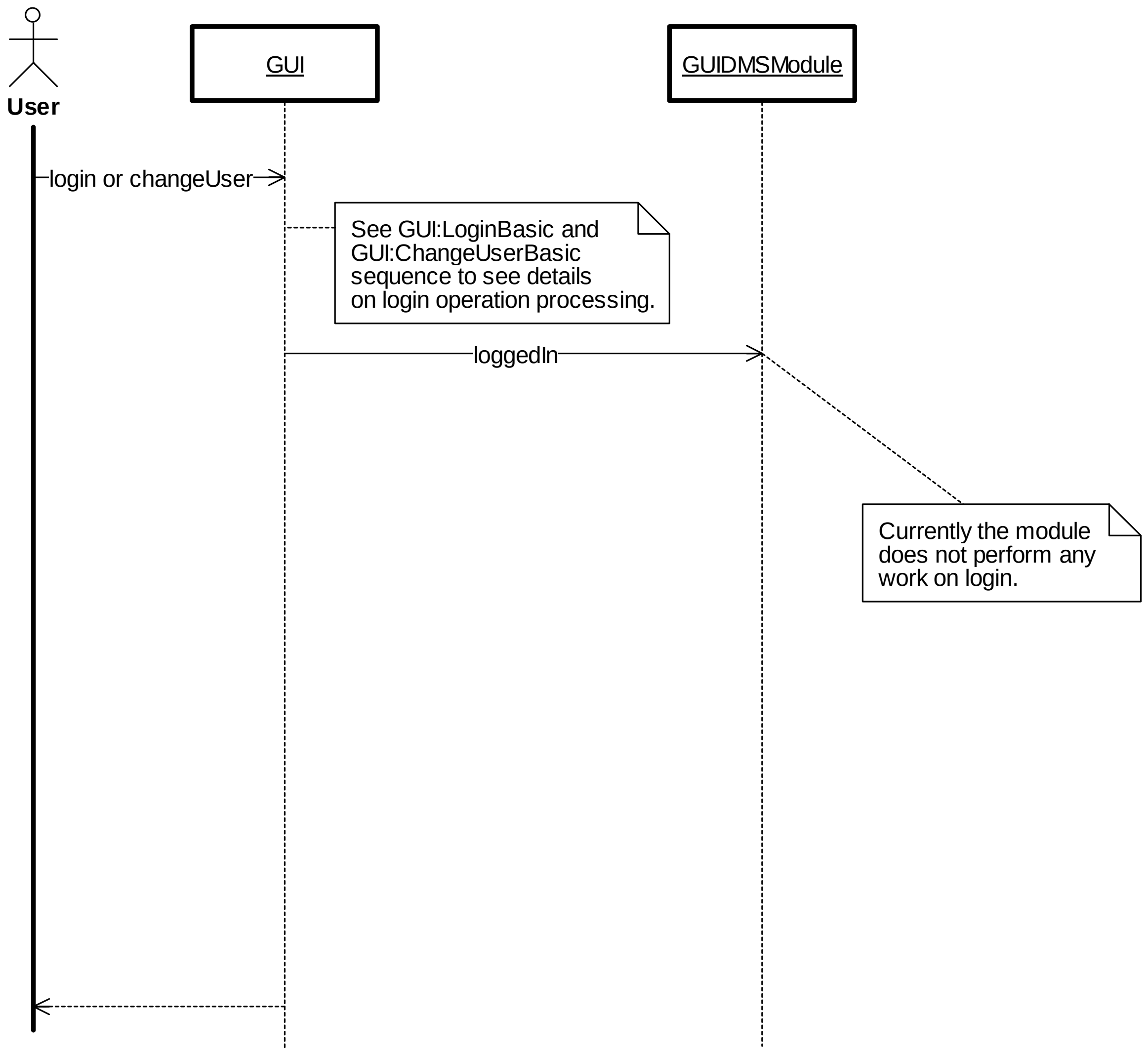

User
GUI
GUIDMSModule
login or changeUser
See GUI:LoginBasic and GUI:ChangeUserBasic sequence to see details on login operation processing.
loggedIn
Currently the module does not perform any work on login.

Figure 37. GUIDMSModule:Login (Sequence Diagram)

3.6.2.11 GUIDMSModule:Logout (Sequence Diagram)

This sequence of events is initiated when a user logs out of the system using either the logout or change user commands from the toolbar window. These commands cause the GUI:LogoutBasic or GUI ChangeUserBasic sequences to be performed. As part of these sequences, the GUI will call each of the installed modules giving them a chance to perform necessary operations to clean up data for a particular user. The GUIDMSModule does not currently need to perform any processing when a user logs out.

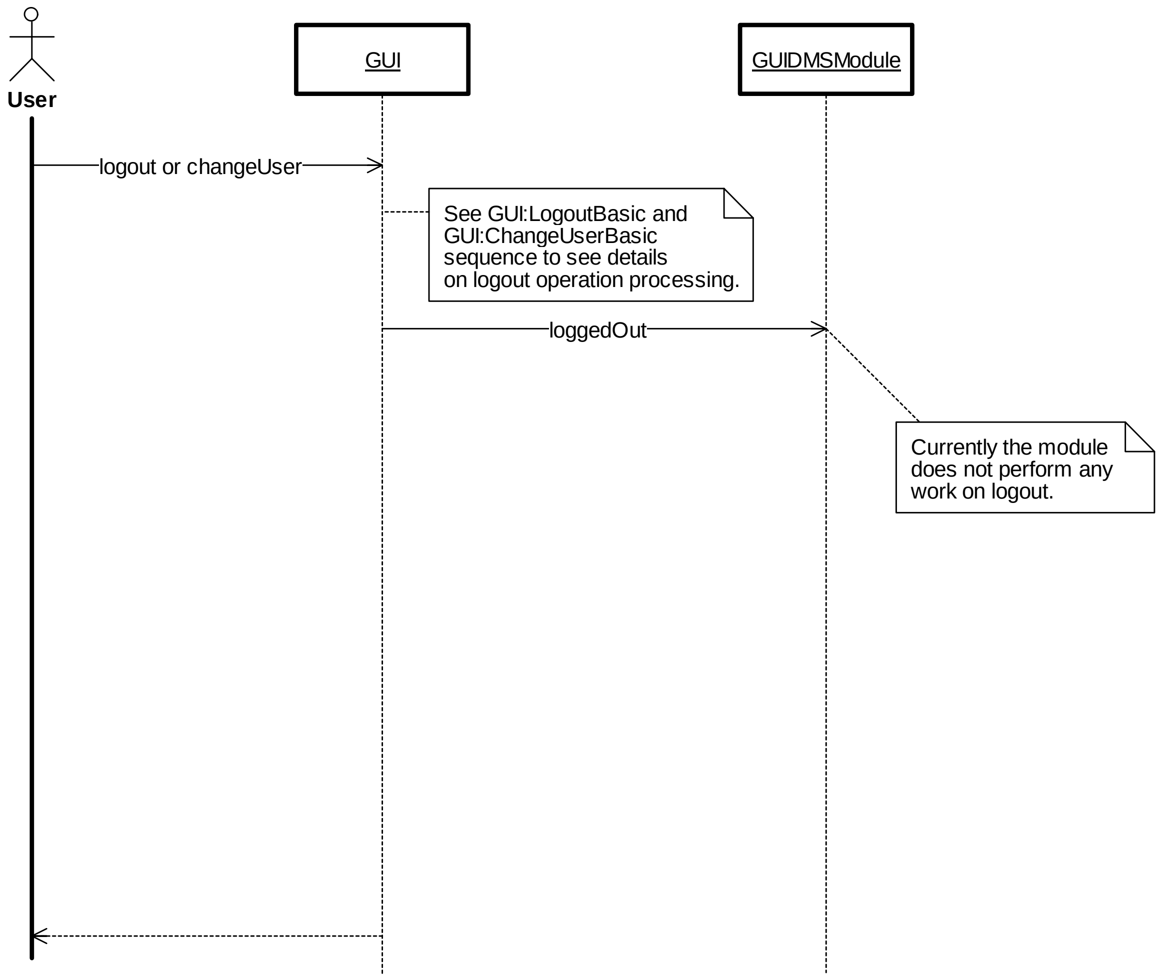

User
GUI
GUIDMSModule
logout or changeUser
See GUI:LogoutBasic and GUI:ChangeUserBasic sequence to see details on logout operation processing.
loggedOut
Currently the module does not perform any work on logout.

Figure 38. GUIDMSModule:Logout (Sequence Diagram)

3.6.2.12 GUIDMSModule:ModifyDMSSettings (Sequence Diagram)

This sequence shows how an operator may alter the configuration of a Default DMS. Refer to the ModifyFP9500Settings sequence diagram for the processing required for a specific DMS model. The operator initiates this action by right clicking on the DMS in a window and selecting the “Properties” menu item. If the user does not have the appropriate functional rights, this menu item will not be made available. The GUIDefaultDMS object creates a DefaultDMSPropertiesDialog, which displays the current DMS configuration and allows the user to modify the current configuration. When the operator is done editing the configuration, clicking on the “OK” button on the dialog causes the GUIDefaultDMS module to create a CommandStatus object, and a DMSConfiguration object and then call the Chart2DMS object to reconfigure itself by calling the setConfiguration method. The setConfiguration returns control immediately and performs the DMS reconfiguration operation asynchronously, barring any user privilege issues. The operation may involve field communication for certain device models. The progress of the command is communicated to the user via the CommandStatus object, which is updated by the server.

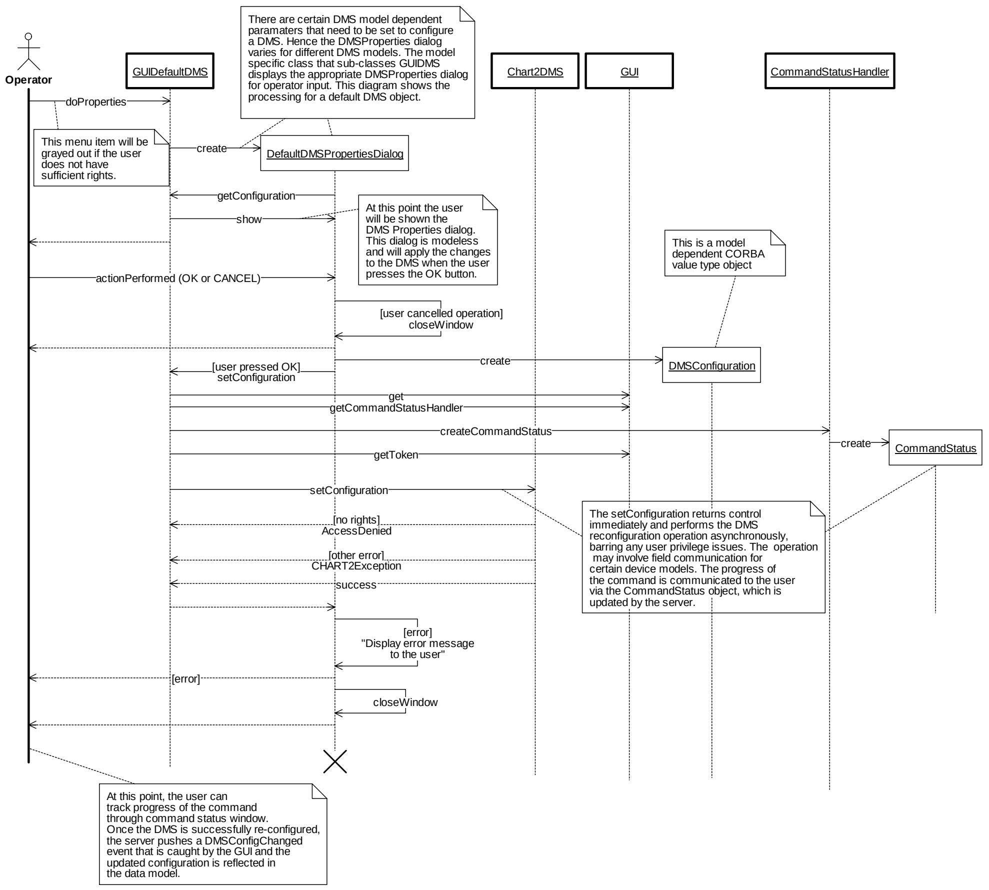
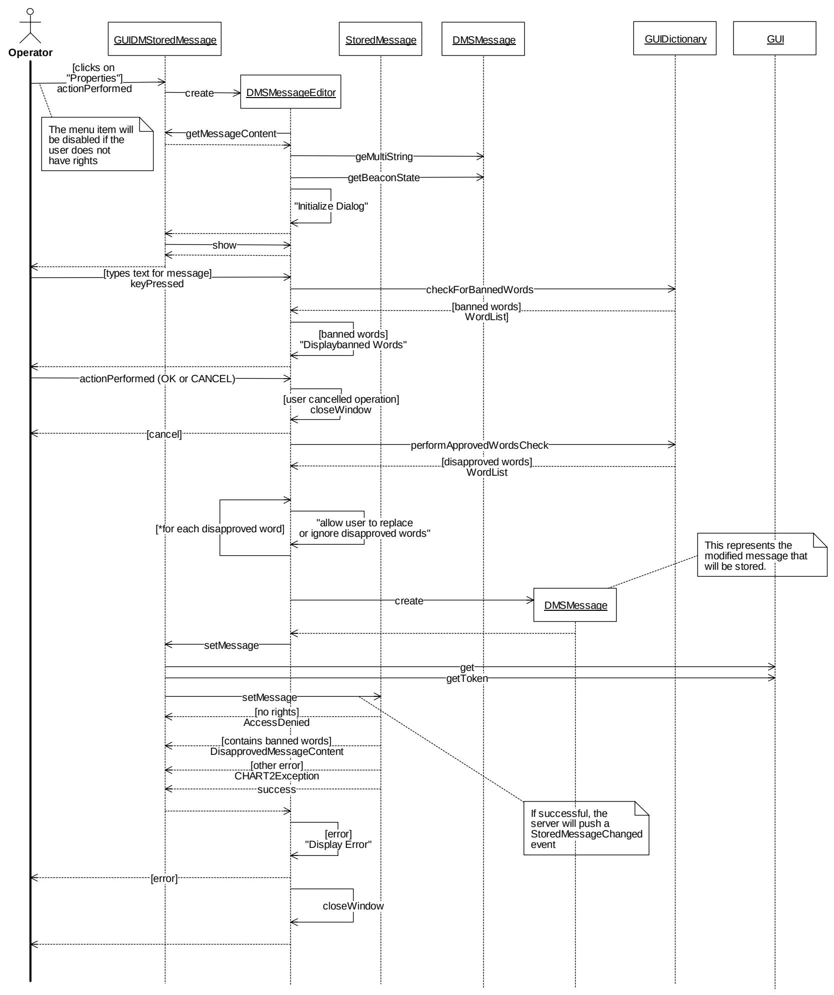

Operator
GUIDefaultDMS
Chart2DMS
GUI
CommandStatusHandler
There are certain DMS model dependent paramaters that need to be set to configure a DMS. Hence the DMSProperties dialog varies for different DMS models. The model specific class that sub-classes GUIDMS displays the appropriate DMSProperties dialog for operator input. This diagram shows the processing for a default DMS object.
doProperties
This menu item will be grayed out if the user does not have sufficient rights.
create
DefaultDMSPropertiesDialog
getConfiguration
show
At this point the user will be shown the DMS Properties dialog. This dialog is modeless and will apply the changes to the DMS when the user presses the OK button.
This is a model dependent CORBA value type object
actionPerformed (OK or CANCEL)
[user cancelled operation] closeWindow
create
DMSConfiguration
[user pressed OK] setConfiguration
get
getCommandStatusHandler
createCommandStatus
create
CommandStatus
getToken
setConfiguration
The setConfiguration returns control immediately and performs the DMS reconfiguration operation asynchronously, barring any user privilege issues. The operation may involve field communication for certain device models. The progress of the command is communicated to the user via the CommandStatus object, which is updated by the server.
[no rights] AccessDenied
[other error] CHART2Exception
success
[error] "Display error message to the user"
[error]
closeWindow
At this point, the user can track progress of the command through command status window. Once the DMS is successfully re-configured, the server pushes a DMSConfigChanged event that is caught by the GUI and the updated configuration is reflected in the data model.

Figure 39. GUIDMSModule:ModifyDMSSettings (Sequence Diagram)

3.6.2.13 GUIDMSModule:ModifyDMSStoredMessage (Sequence Diagram)

This diagram shows how the contents of a stored message are modified. The user clicks on an existing GUIDMSStoredMessage object, and clicks on the “Properties” menu item. The GUIDMSStoredMessage then invokes the DMSMessageEditor dialog. On initialization, the dialog calls back to the GUIDMSStoredMessage wrapper object to get the message content, which calls back to the StoredMessage object in the server, if necessary. When the DMSMessage is returned, the dialog can be initialized from the existing message contents. As the user types in text for the message, banned words will be displayed. When the user clicks OK, the message editor will check for disapproved words and provides suggestions to replace these words. A new DMSMessage object is created with the user modifications. The GUIDMSStoredMessage is called to set the message, which in turn calls the StoredMessage object in the server. If successful, the server will push a CORBA event to update the clients.



Operator
GUIDMSStoredMessage
StoredMessage
DMSMessage
GUIDictionary
GUI
[clicks on "Properties"] actionPerformed
create
DMSMessageEditor
The menu item will be disabled if the user does not have rights
getMessageContent
geMultiString
getBeaconState
"Initialize Dialog"
show
[types text for message] keyPressed
checkForBannedWords
[banned words] WordList
[banned words] "Displaybanned Words"
actionPerformed (OK or CANCEL)
[user cancelled operation] closeWindow
[cancel]
performApprovedWordsCheck
[disapproved words] WordList
[*for each disapproved word]
"allow user to replace or ignore disapproved words"
This represents the modified message that will be stored.
create
DMSMessage
setMessage
get
getToken
setMessage
[no rights] AccessDenied
[contains banned words] DisapprovedMessageContent
[other error] CHART2Exception
success
If successful, the server will push a StoredMessageChanged event
[error] "Display Error"
[error]
closeWindow

Figure 40. GUIDMSModule:ModifyDMSStoredMessage (Sequence Diagram)

3.6.2.14 GUIDMSModule:PollNow (Sequence Diagram)

This sequence diagram shows how a user with appropriate rights can perform a forced polling of a DMS, when the device is in maintenance mode. The sequence is initiated when the user right clicks on a GUIDMS object and selects the "Poll Now" menu item. The GUIDMS object creates a CommandStatus object and then calls the Chart2DMS object that it wraps to perform the operation. The progress of the poll now operation is displayed to the user on the command status window, which is updated as the server updates the CommandStatus object that was passed to it along with the poll command. If successful, the server will push CORBA events indicating the changed DMS status.

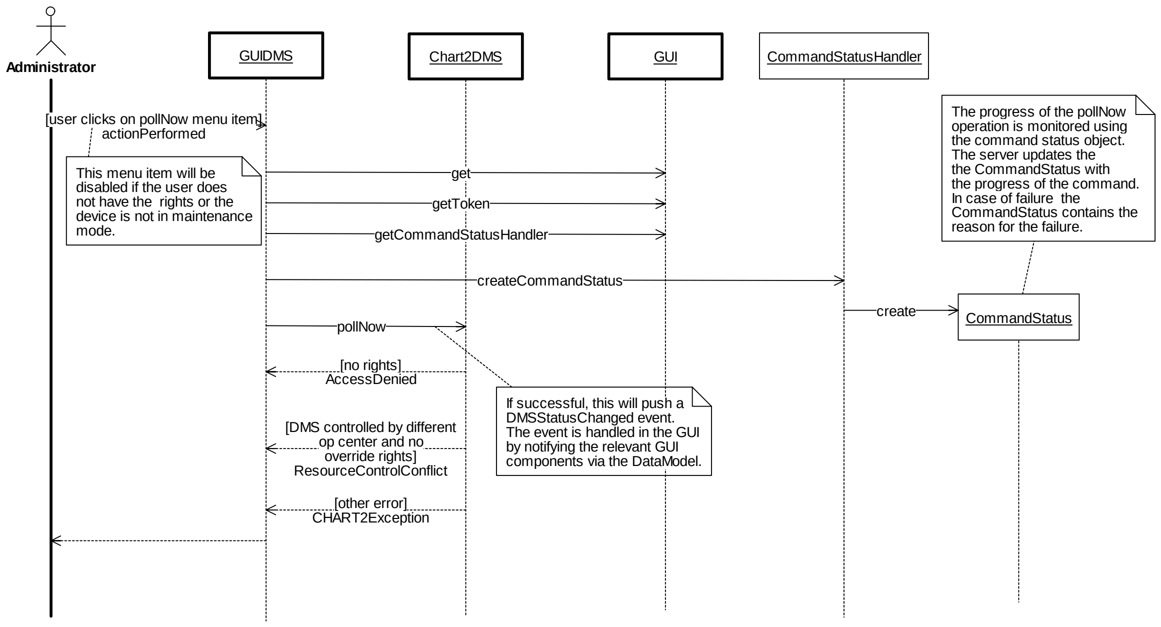

Administrator
GUIDMS
Chart2DMS
GUI
CommandStatusHandler
[user clicks on pollNow menu item] actionPerformed
This menu item will be disabled if the user does not have the rights or the device is not in maintenance mode.
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
The progress of the pollNow operation is monitored using the command status object. The server updates the the CommandStatus with the progress of the command. In case of failure the CommandStatus contains the reason for the failure.
pollNow
[no rights] AccessDenied
If successful, this will push a DMSStatusChanged event. The event is handled in the GUI by notifying the relevant GUI components via the DataModel.
[DMS controlled by different op center and no override rights] ResourceControlConflict
[other error] CHART2Exception

Figure 41. GUIDMSModule:PollNow (Sequence Diagram)

3.6.2.15 GUIDMSModule:PutDMSInMaintenanceMode (Sequence Diagram)

This diagram shows how a DMS is put into maintenance mode. The Administrator right clicks on a GUIDMS and clicks on the “Put In Maintenance Mode” menu item. The GUIDMS creates a CommandStatus object to monitor the progress of the command and calls the CHART2DMS object (which it wraps) to put it in maintenance mode. If successful, the server will push a CORBA event indicating that the comm mode has been changed.

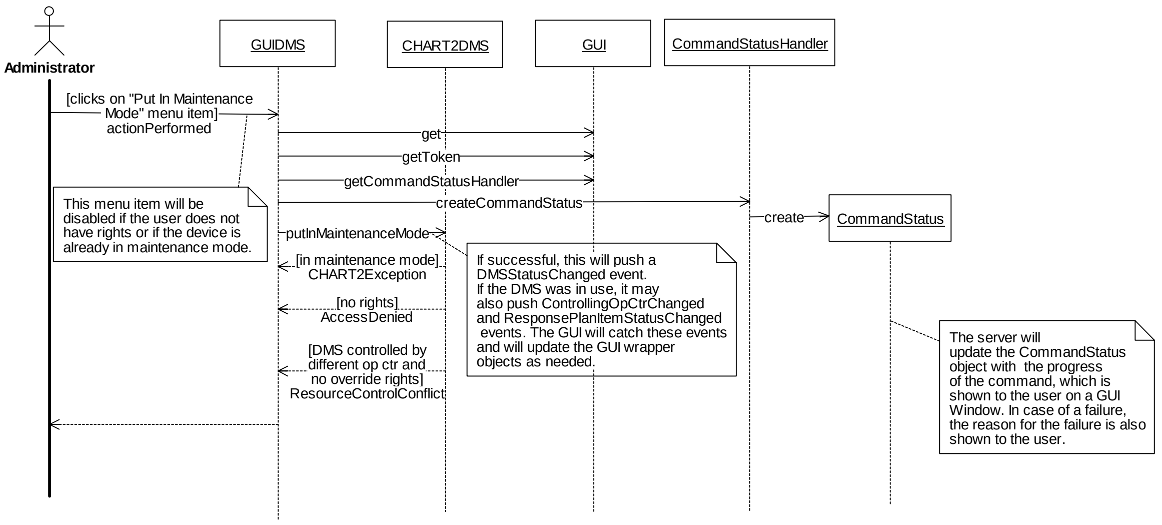

Administrator
GUIDMS
CHART2DMS
GUI
CommandStatusHandler
[clicks on "Put In Maintenance Mode" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is already in maintenance mode.
putInMaintenanceMode
[in maintenance mode] CHART2Exception
[no rights] AccessDenied
[DMS controlled by different op ctr and no override rights] ResourceControlConflict
If successful, this will push a DMSStatusChanged event. If the DMS was in use, it may also push ControllingOpCtrChanged and ResponsePlanItemStatusChanged events. The GUI will catch these events and will update the GUI wrapper objects as needed.
The server will update the CommandStatus object with the progress of the command, which is shown to the user on a GUI Window. In case of a failure, the reason for the failure is also shown to the user.

Figure 42. GUIDMSModule:PutDMSInMaintenanceMode (Sequence Diagram)

3.6.2.16 GUIDMSModule:PutOnline (Sequence Diagram)

This diagram shows how a DMS is put online. The Administrator right clicks on a GUIDMS and clicks on the “Put Online” menu item. The GUIDMS creates a CommandStatus object to monitor the progress of the command and calls the CHART2DMS object (which it wraps) to put it online. If successful, the server will push a CORBA event indicating that the comm mode has been changed.

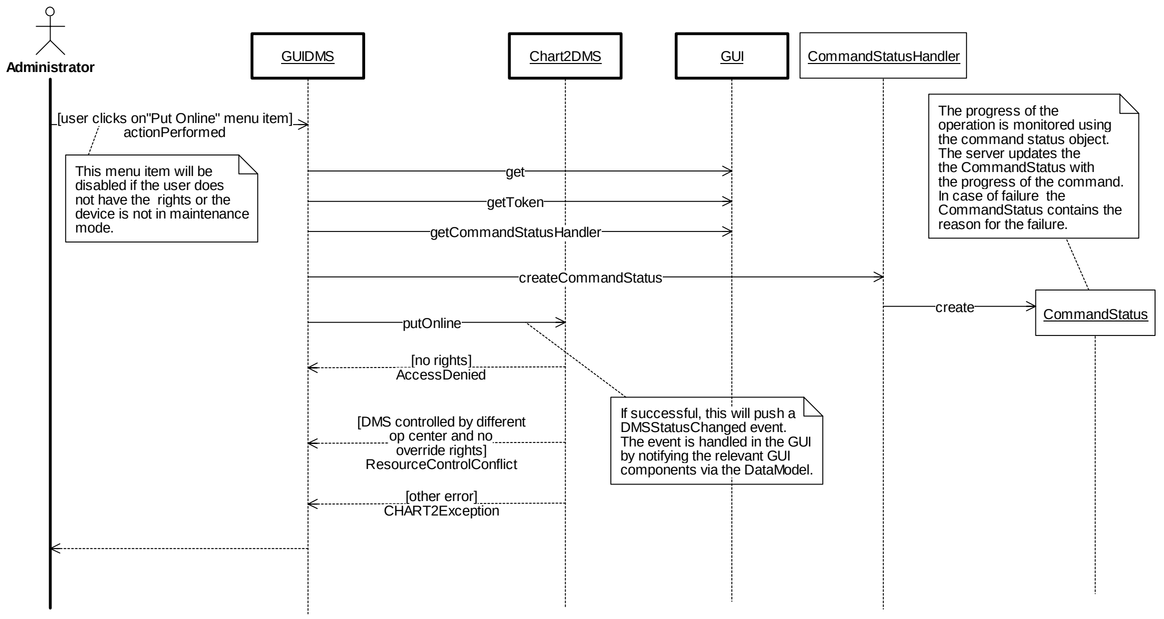

Administrator
GUIDMS
Chart2DMS
GUI
CommandStatusHandler
CommandStatus
[user clicks on"Put Online" menu item]
actionPerformed
This menu item will be disabled if the user does not have the rights or the device is not in maintenance mode.
get
getToken
getCommandStatusHandler
createCommandStatus
create
putOnline
[no rights]
AccessDenied
[DMS controlled by different op center and no override rights]
ResourceControlConflict
[other error]
CHART2Exception
The progress of the operation is monitored using the command status object. The server updates the the CommandStatus with the progress of the command. In case of failure the CommandStatus contains the reason for the failure.
If successful, this will push a DMSStatusChanged event. The event is handled in the GUI by notifying the relevant GUI components via the DataModel.

Figure 43. GUIDMSModule:PutOnline (Sequence Diagram)

3.6.2.17 GUIDMSModule:RemoveDMS (Sequence Diagram)

This diagram shows how a DMS is removed from the system. The Administrator right clicks on a GUIDMS object and clicks on the “Remove” menu item. The GUIDMS creates a CommandStatus object to monitor the progress of the command and calls the remove() method of the Chart2DMS object (which it wraps). If successful, the server will push a CORBA event indicating that the DMS was removed.

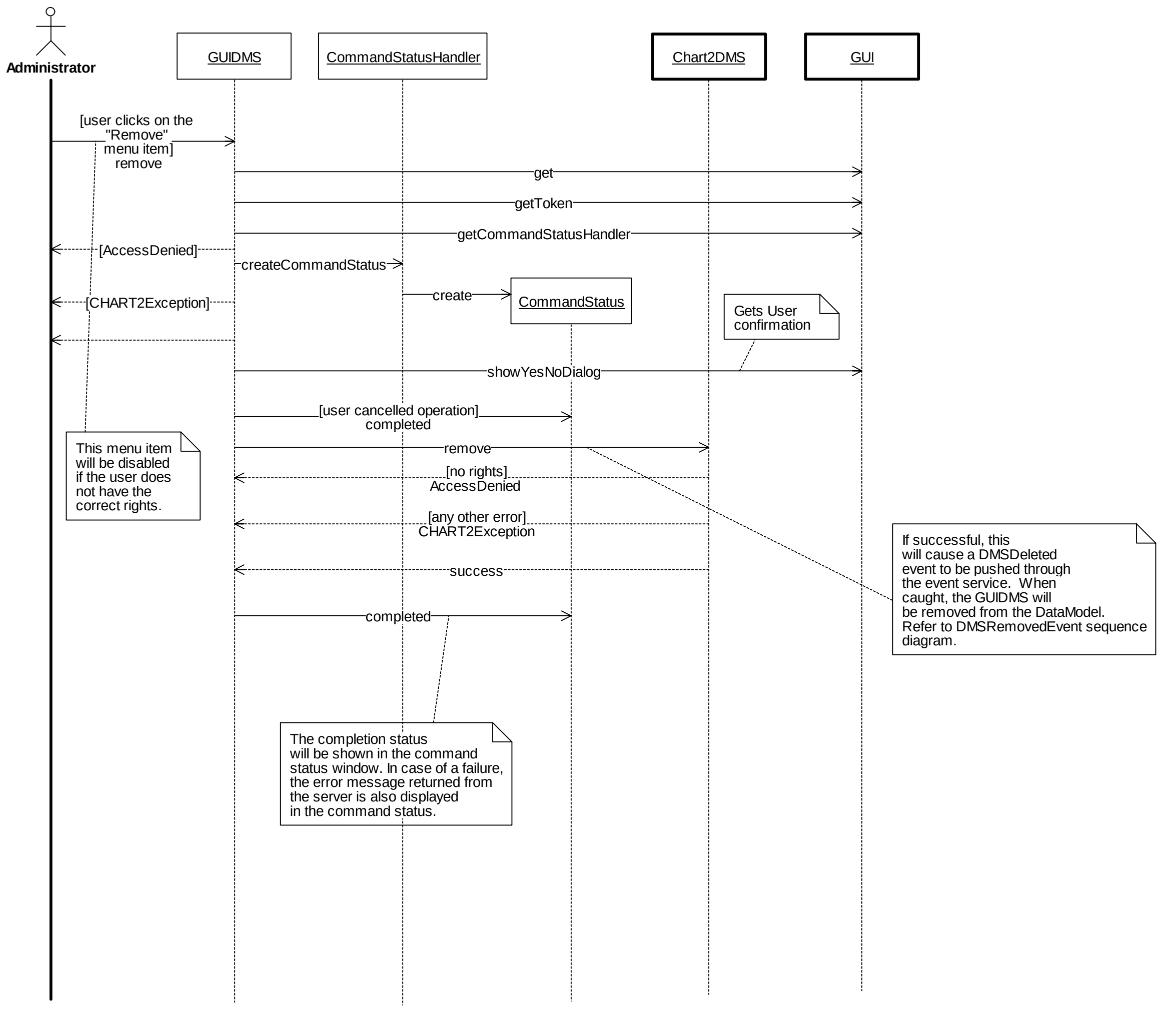

Administrator
GUIDMS
CommandStatusHandler
Chart2DMS
GUI
[user clicks on the "Remove" menu item] remove
get
getToken
getCommandStatusHandler
[AccessDenied]
createCommandStatus
create
CommandStatus
[CHART2Exception]
Gets User confirmation
showYesNoDialog
[user cancelled operation] completed
This menu item will be disabled if the user does not have the correct rights.
remove
[no rights] AccessDenied
[any other error] CHART2Exception
success
completed
If successful, this will cause a DMSDeleted event to be pushed through the event service. When caught, the GUIDMS will be removed from the DataModel. Refer to DMSRemovedEvent sequence diagram.
The completion status will be shown in the command status window. In case of a failure, the error message returned from the server is also displayed in the command status.

Figure 44. GUIDMSModule:RemoveDMS (Sequence Diagram)

3.6.2.18 GUIDMSModule:Reset (Sequence Diagram)

This sequence diagram shows how a user with appropriate rights can reset a DMS, when the device is in maintenance mode. The sequence is initiated when the user right clicks on a GUIDMS object and selects the “Reset” menu item. The GUIDMS object creates a CommandStatus object and then calls the Chart2DMS object that it wraps to perform the operation. The progress of the reset operation is displayed to the user on the command status window, which is updated as the server updates the CommandStatus object that was passed to it along with the reset command. If successful, the server will push a CORBA event indicating the changed DMS status.

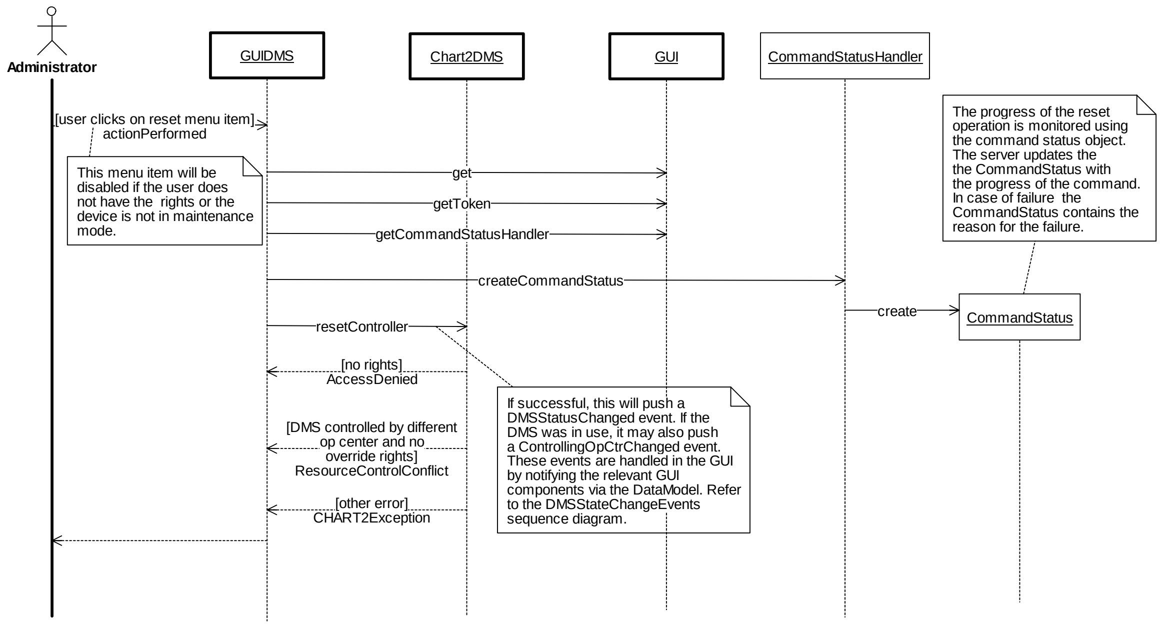

Administrator
GUIDMS
Chart2DMS
GUI
CommandStatusHandler
[user clicks on reset menu item] actionPerformed
This menu item will be disabled if the user does not have the rights or the device is not in maintenance mode.
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
The progress of the reset operation is monitored using the command status object. The server updates the the CommandStatus with the progress of the command. In case of failure the CommandStatus contains the reason for the failure.
resetController
[no rights] AccessDenied
[DMS controlled by different op center and no override rights] ResourceControlConflict
[other error] CHART2Exception
If successful, this will push a DMSStatusChanged event. If the DMS was in use, it may also push a ControllingOpCtrChanged event. These events are handled in the GUI by notifying the relevant GUI components via the DataModel. Refer to the DMSStateChangeEvents sequence diagram.

Figure 45. GUIDMSModule:Reset (Sequence Diagram)

3.6.2.19 GUIDMSModule:SetMessageInMaintenanceMode (Sequence Diagram)

This shows how a message is set on a DMS when it is in maintenance mode. The user right clicks on the GUIDMS object and clicks on the “Edit Message (Auto)” or “Edit Message (Manual)” menu item. The GUIDMS object invokes the DMSMessageEditor dialog. The DMSMessageEditor dialog is initially populated with the current message displayed on the DMS, if any. The user may use this dialog to type in a new message and preview what that message will look like formatted for the selected DMS. As the user types a text message, the banned words are displayed. When the user clicks OK, the message editor will check for disapproved words and provides suggestions to replace these words. A DMSMessage object is created and GUIDMS is called to set the message. The GUIDMS object creates a CommandStatus to monitor the progress of the command, then calls the CHART2DMS object which it wraps. If successful, the server will push CORBA events to update the clients for any state changes.

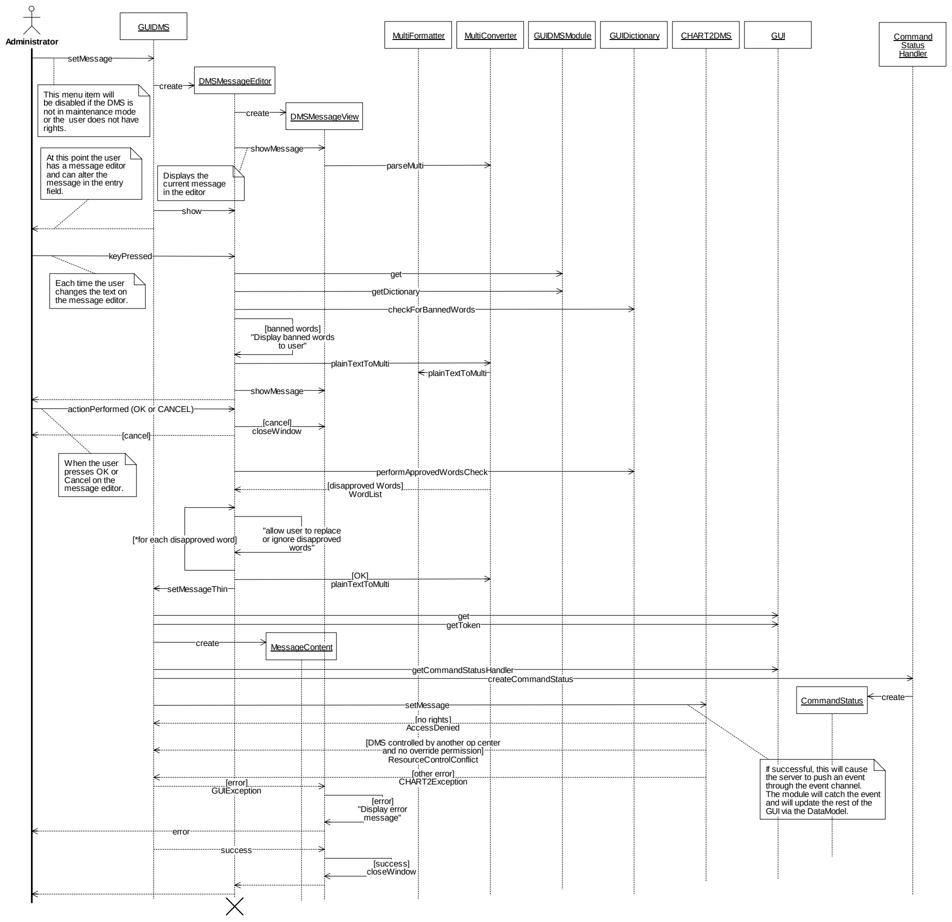

Administrator
GUIDMS
MultiFormatter
MultiConverter
GUIDMSModule
GUIDictionary
CHART2DMS
GUI
Command Status Handler
setMessage
create
DMSMessageEditor
This menu item will be disabled if the DMS is not in maintenance mode or the user does not have rights.
create
DMSMessageView
showMessage
parseMulti
At this point the user has a message editor and can alter the message in the entry field.
Displays the current message in the editor
show
keyPressed
Each time the user changes the text on the message editor.
get
getDictionary
checkForBannedWords
[banned words] "Display banned words to user"
plainTextToMulti
plainTextToMulti
showMessage
actionPerformed (OK or CANCEL)
[cancel] closeWindow
[cancel]
When the user presses OK or Cancel on the message editor.
performApprovedWordsCheck
[disapproved Words] WordList
[*for each disapproved word]
"allow user to replace or ignore disapproved words"
[OK] plainTextToMulti
setMessageThin
get
getToken
create
MessageContent
getCommandStatusHandler
createCommandStatus
CommandStatus
create
setMessage
[no rights] AccessDenied
[DMS controlled by another op center and no override permission] ResourceControlConflict
[other error] CHART2Exception
[error] GUIException
If successful, this will cause the server to push an event through the event channel. The module will catch the event and will update the rest of the GUI via the DataModel.
[error] "Display error message"
error
success
[success] closeWindow

Figure 46. GUIDMSModule:SetMessageInMaintenanceMode (Sequence Diagram)

3.6.2.20 GUIDMSModule:ShowTrueDisplay (Sequence Diagram)

This sequence shows how an operator may view the current message displayed on a particular DMS. The view will be formatted to show the message as it looks on the sign. The operator initiates this sequence by right clicking on the desired DMS in a window and selecting the "Show Display" menu item.

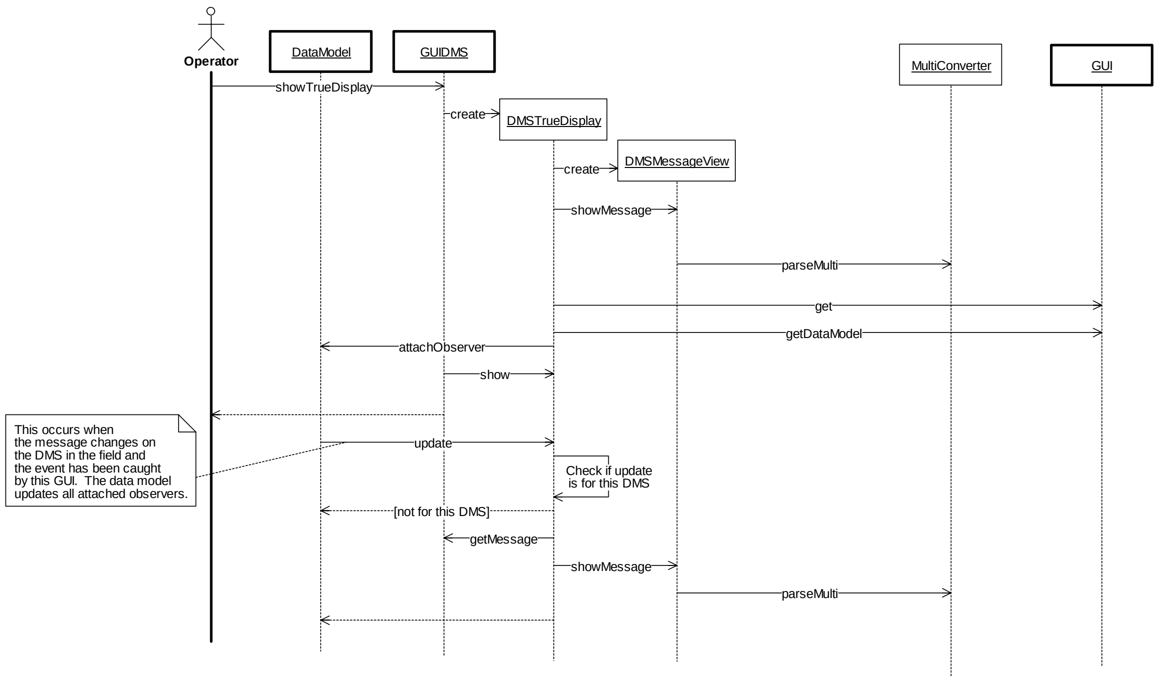


Figure 47. GUIDMSModule:ShowTrueDisplay (Sequence Diagram)

3.6.2.21 GUIDMSModule:Shutdown (Sequence Diagram)

This diagram shows processing involved in the shutdown of the GUIDMSModule. At the time of GUI shutdown all of the installable modules including the GUIDMSModule is called on to perform cleanup operations by calling their shutdown method. On shutdown, the GUIDMSModule disconnects itself from the ORB.

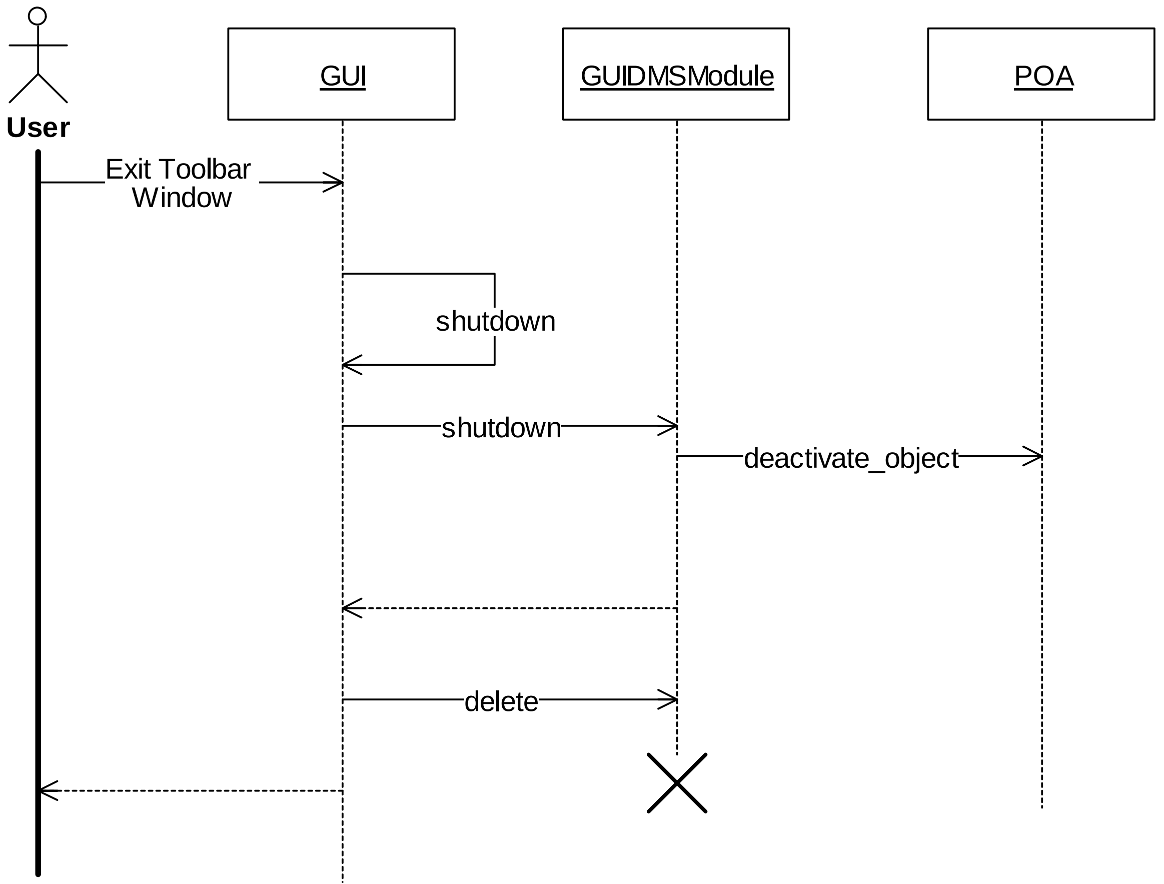


Figure 48. GUIDMSModule:Shutdown (Sequence Diagram)

3.6.2.22 GUIDMSModule:Startup (Sequence Diagram)

This diagram shows the processing involved during startup of the GUIDMS module. At GUI startup time, each of the installable modules including the GUIDMSModule is initialized by calling their startup routines. The GUIDMSModule connects itself to the ORB in order to receive DMS related CORBA events from the event service. A DMSNavGroup object is created to manage the GUIDMS related objects in the GUI navigator. A GUIDMSModelFactory object is created which will aid the GUIDMSModule in the creation of model specific GUIDMS objects. Finally, the GUIDMSModule registers itself with the GUIPlanModule, GUILibraryModule and the GUITrafficEventModule in order to be able to support GUIPlanItem, GUIDMSStoredMessage and GUIDMSResponsePlanItem objects respectively.

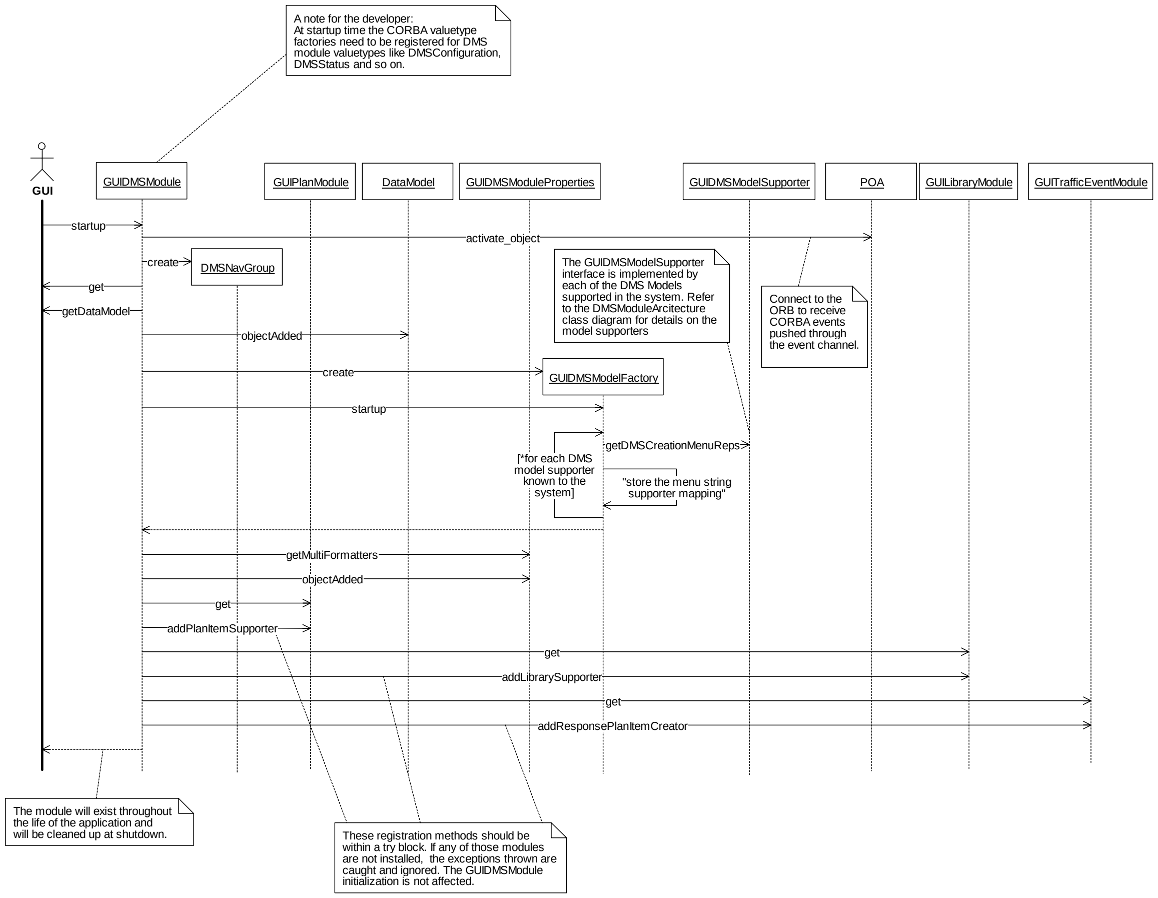

A note for the developer: At startup time the CORBA valuetype factories need to be registered for DMS module valuetypes like DMSConfiguration, DMSStatus and so on.
GUI
GUIDMSModule
GUIPlanModule
DataModel
GUIDMSModuleProperties
GUIDMSModelSupporter
POA
GUILibraryModule
GUITrafficEventModule
startup
activate_object
create
DMSNavGroup
get
getDataModel
The GUIDMSModelSupporter interface is implemented by each of the DMS Models supported in the system. Refer to the DMSModuleArcitecture class diagram for details on the model supporters
Connect to the ORB to receive CORBA events pushed through the event channel.
objectAdded
create
GUIDMSModelFactory
startup
getDMSCreationMenuReps
[*for each DMS model supporter known to the system]
"store the menu string supporter mapping"
getMultiFormatters
objectAdded
get
addPlanItemSupporter
get
addLibrarySupporter
get
addResponsePlanItemCreator
The module will exist throughout the life of the application and will be cleaned up at shutdown.
These registration methods should be within a try block. If any of those modules are not installed, the exceptions thrown are caught and ignored. The GUIDMSModule initialization is not affected.

Figure 49. GUIDMSModule:Startup (Sequence Diagram)

3.6.2.23 GUIDMSModule:TakeOffline (Sequence Diagram)

This sequence diagram shows how an operator with the appropriate rights can take a DMS offline. The sequence is initiated when the user right clicks on a GUIDMS object in the navigator and selects the “Take Offline” menu item. The GUIDMS creates a CommandStatus object and calls the CHART2DMS object (that it wraps) to execute the offline command. The progress of the operation is displayed to the user on the command status window, which is updated as the server updates the CommandStatus object that was passed to it along with the offline command. If successful, the server will push a CORBA event indicating that the DMS has been taken offline.

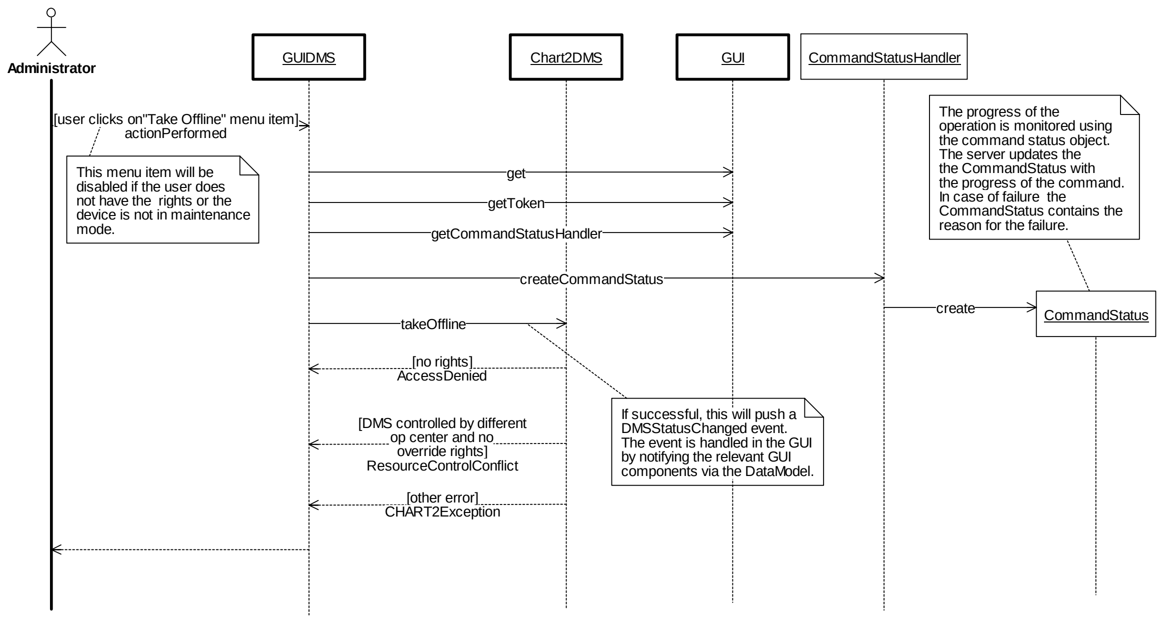

Administrator
GUIDMS
Chart2DMS
GUI
CommandStatusHandler
[user clicks on"Take Offline" menu item] actionPerformed
This menu item will be disabled if the user does not have the rights or the device is not in maintenance mode.
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
The progress of the operation is monitored using the command status object. The server updates the the CommandStatus with the progress of the command. In case of failure the CommandStatus contains the reason for the failure.
takeOffline
[no rights] AccessDenied
[DMS controlled by different op center and no override rights] ResourceControlConflict
If successful, this will push a DMSStatusChanged event. The event is handled in the GUI by notifying the relevant GUI components via the DataModel.
[other error] CHART2Exception

Figure 50 GUIDMSModule:TakeOffline (Sequence Diagram)

3.6.2.24 GUIDMSModule:TrafficEventResponse-BlankDMS (Sequence Diagram)

This diagram shows how the closing of a traffic event blanks a DMS that was earlier used to display a message in response to the event. A DMS may also be blanked when the response item associated to this DMS is removed from the traffic event. The user right clicks on the GUIDMSResponsePlanItem or the GUITrafficEvent objects and selects the appropriate menu item. In both cases, the remove method of the GUIDMSResponsePlanItem wrapper object will be called, which will in turn call the served ResponsePlanItem object that it wraps. If successful, the server will push events to all GUIs indicating the changed status.

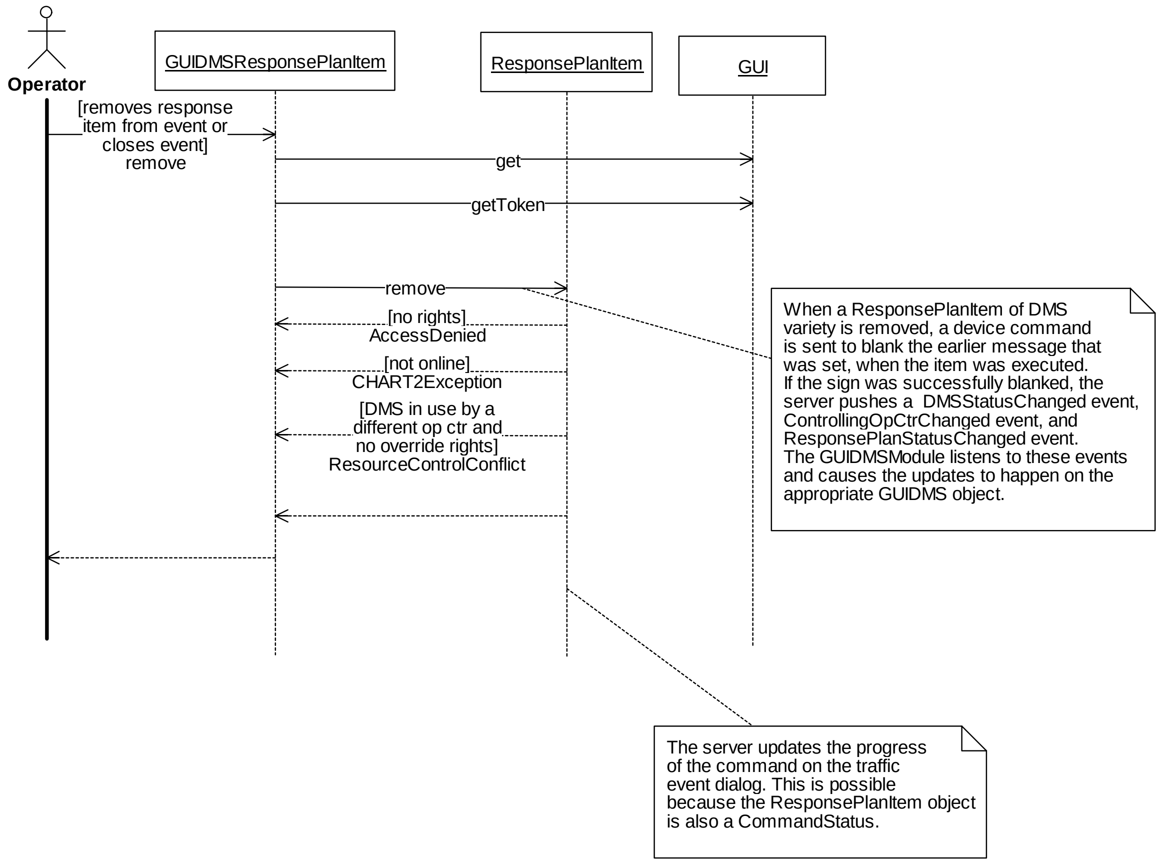

Operator
GUIDMSResponsePlanItem
ResponsePlanItem
GUI
[removes response item from event or closes event] remove
get
getToken
remove
[no rights] AccessDenied
[not online] CHART2Exception
[DMS in use by a different op ctr and no override rights] ResourceControlConflict
When a ResponsePlanItem of DMS variety is removed, a device command is sent to blank the earlier message that was set, when the item was executed. If the sign was successfully blanked, the server pushes a DMSStatusChanged event, ControllingOpCtrChanged event, and ResponsePlanStatusChanged event. The GUIDMSModule listens to these events and causes the updates to happen on the appropriate GUIDMS object.
The server updates the progress of the command on the traffic event dialog. This is possible because the ResponsePlanItem object is also a CommandStatus.

Figure 51. GUIDMSModule:TrafficEventResponse-BlankDMS (Sequence Diagram)

3.6.2.25 GUIDMSModule:ModifyFP9500Settings (Sequence Diagram)

This sequence shows how an operator may alter the configuration of a FP9500 DMS. In fact the processing shown here will be the same for any other model of DMS in the system. Refer to the ModifyDMSSettings sequence diagram for the processing required for a default DMS. The operator initiates this action by right clicking on the DMS in a window and selecting the “Properties” menu item. If the user does not have the appropriate functional rights, this menu item will not be made available. The GUIFP9500 object creates a FP9500PropertiesDialog, which displays the current DMS configuration and allows the user to modify the current configuration. When the operator is done editing the configuration, clicking on the “OK” button on the dialog causes the GUIFP9500 module to create a CommandStatus object, and a DMSConfiguration object and then call the Chart2DMS object to reconfigure itself by calling the setConfiguration method. The setConfiguration return control immediately and performs the DMS reconfiguration operation asynchronously, barring any user privilege issues. The operation may involve field communication for certain device models. The progress of the command is communicated to the user via the CommandStatus object, which is updated by the server.

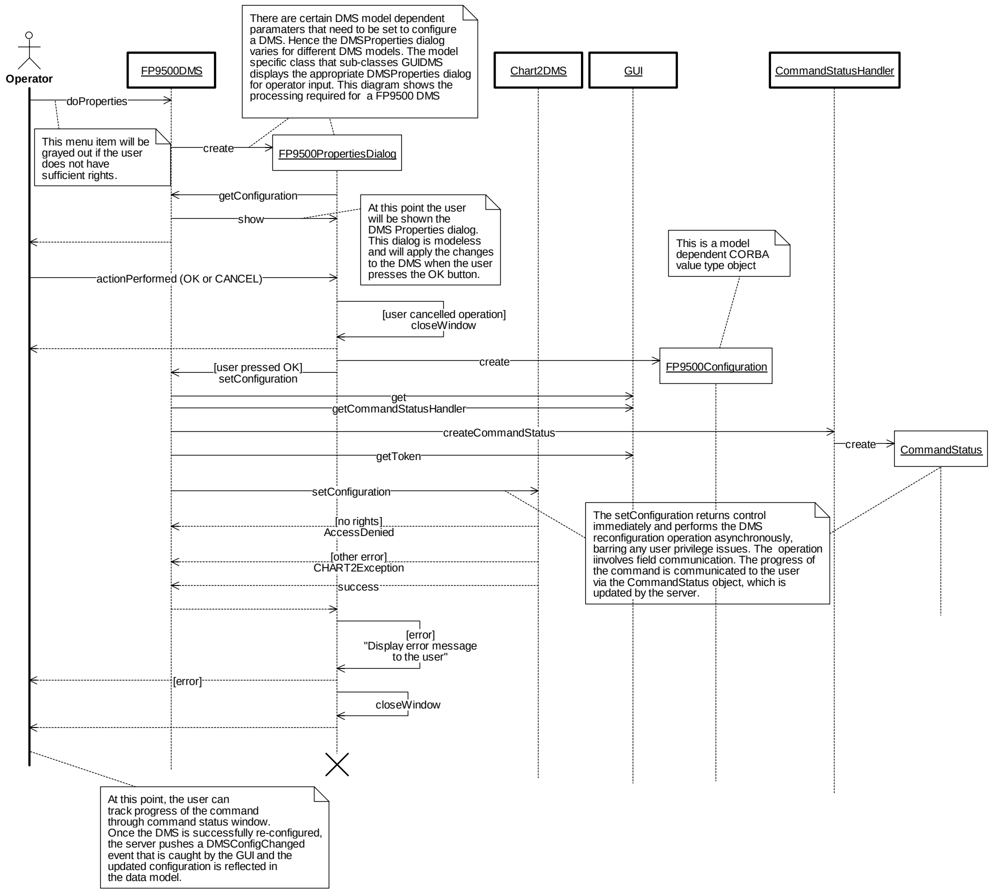

Operator
FP9500DMS
There are certain DMS model dependent paramaters that need to be set to configure a DMS. Hence the DMSProperties dialog varies for different DMS models. The model specific class that sub-classes GUIDMS displays the appropriate DMSProperties dialog for operator input. This diagram shows the processing required for a FP9500 DMS
Chart2DMS
GUI
CommandStatusHandler
doProperties
This menu item will be grayed out if the user does not have sufficient rights.
create
FP9500PropertiesDialog
getConfiguration
show
At this point the user will be shown the DMS Properties dialog. This dialog is modeless and will apply the changes to the DMS when the user presses the OK button.
This is a model dependent CORBA value type object
actionPerformed (OK or CANCEL)
[user cancelled operation] closeWindow
create
FP9500Configuration
[user pressed OK] setConfiguration
get
getCommandStatusHandler
createCommandStatus
create
CommandStatus
getToken
setConfiguration
The setConfiguration returns control immediately and performs the DMS reconfiguration operation asynchronously, barring any user privilege issues. The operation iinvolves field communication. The progress of the command is communicated to the user via the CommandStatus object, which is updated by the server.
[no rights] AccessDenied
[other error] CHART2Exception
success
[error] "Display error message to the user"
[error]
closeWindow
At this point, the user can track progress of the command through command status window. Once the DMS is successfully re-configured, the server pushes a DMSConfigChanged event that is caught by the GUI and the updated configuration is reflected in the data model.

Figure 52. GUIDMSModule:ModifyFP9500Settings (Sequence Diagram)

3.6.2.26 GUIDMSModule:TrafficEventResponse-SetDMSMessage (Sequence Diagram)

This diagram shows how a message is set on a DMS in response to a traffic event. The operator right clicks on a GUIDMSResponsePlanItem object and clicks on the “Execute” menu item. The GUIDMSResponsePlanItem calls the execute() method of the ResponsePlanItem object that it wraps. If successful, the server will push CORBA events indicating the changes to the state of the DMS. The server will also push events to keep the GUIs updated with the current status of the command.

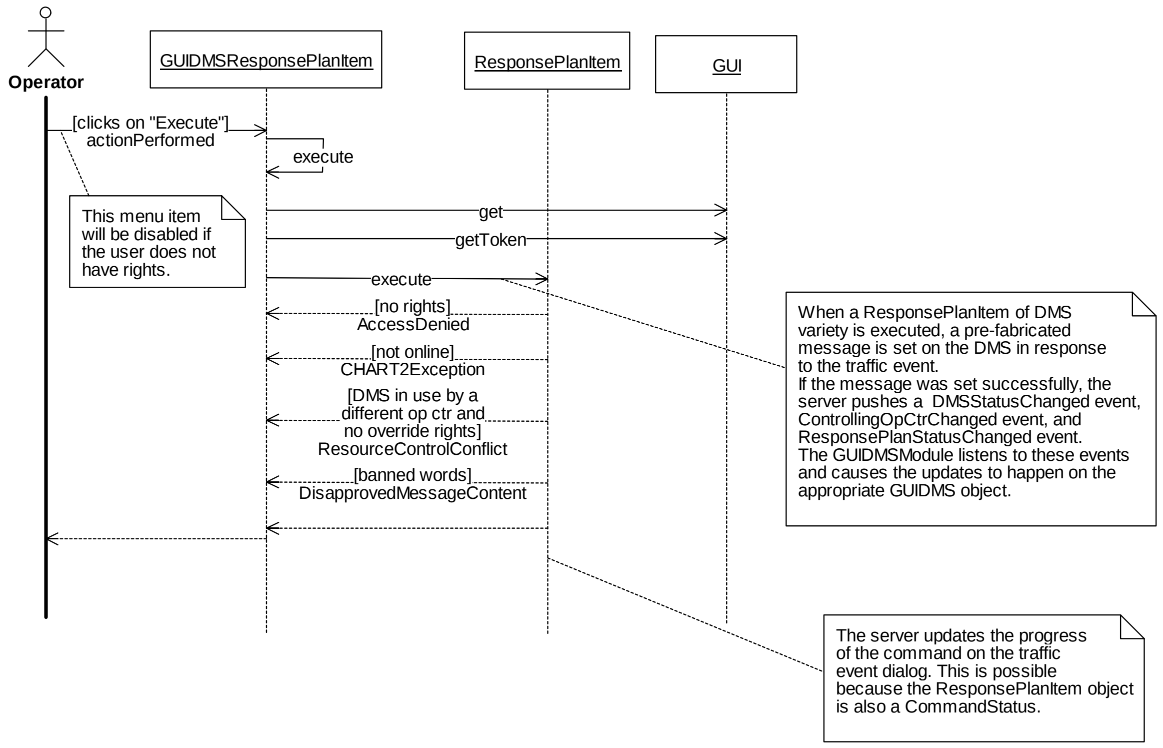

Operator
GUIDMSResponsePlanItem
ResponsePlanItem
GUI
[clicks on "Execute"] actionPerformed
execute
get
getToken
execute
[no rights] AccessDenied
[not online] CHART2Exception
[DMS in use by a different op ctr and no override rights] ResourceControlConflict
[banned words] DisapprovedMessageContent
This menu item will be disabled if the user does not have rights.
When a ResponsePlanItem of DMS variety is executed, a pre-fabricated message is set on the DMS in response to the traffic event. If the message was set successfully, the server pushes a DMSStatusChanged event, ControllingOpCtrChanged event, and ResponsePlanStatusChanged event. The GUIDMSModule listens to these events and causes the updates to happen on the appropriate GUIDMS object.
The server updates the progress of the command on the traffic event dialog. This is possible because the ResponsePlanItem object is also a CommandStatus.

Figure 53. GUIDMSModule:TrafficEventResponse-SetDMSMessage (Sequence Diagram)

3.7 GUIDictionaryModule

3.7.1 Class Diagrams

3.7.1.1 GUIDictionaryModuleClasses (Class Diagram)

This diagram shows the data hierarchy of the GUIDictionaryModule and the objects it supports.

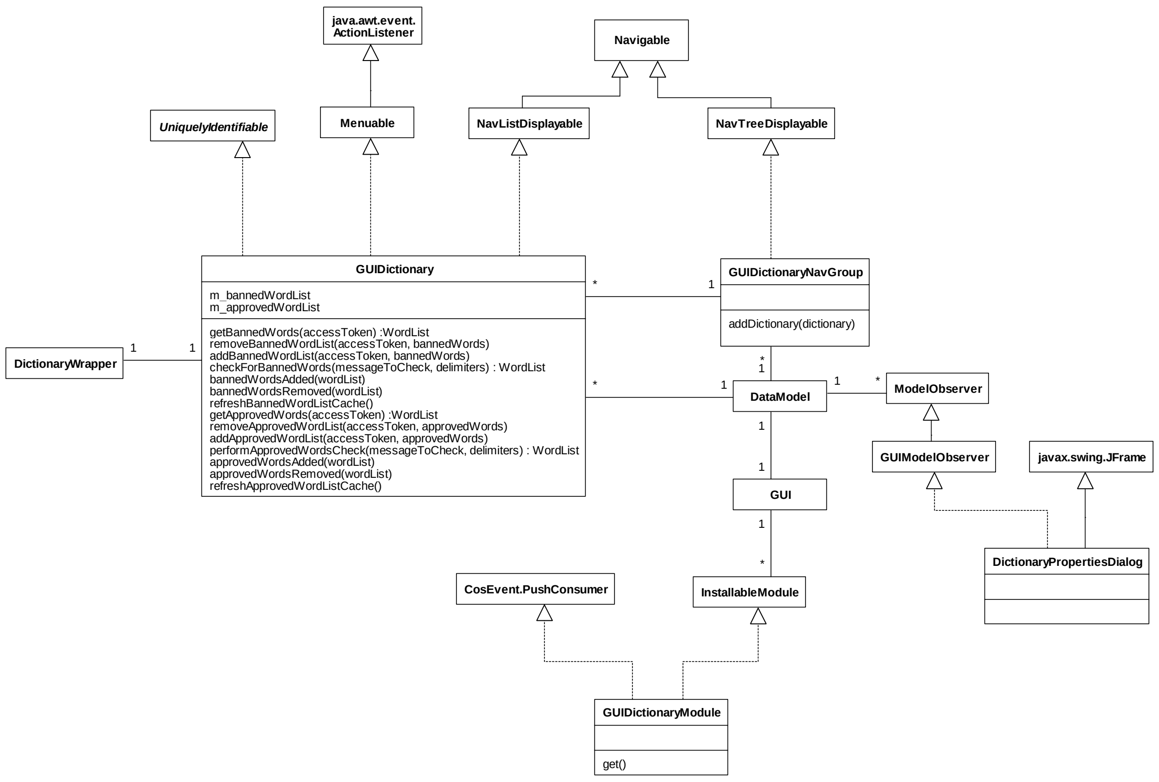

java.awt.event.
ActionListener
Navigable
UniquelyIdentifiable
Menuable
NavListDisplayable
NavTreeDisplayable
GUIDictionary
m_bannedWordList
m_approvedWordList
getBannedWords(accessToken) :WordList
removeBannedWordList(accessToken, bannedWords)
addBannedWordList(accessToken, bannedWords)
checkForBannedWords(messageToCheck, delimiters) : WordList
bannedWordsAdded(wordList)
bannedWordsRemoved(wordList)
refreshBannedWordListCache()
getApprovedWords(accessToken) :WordList
removeApprovedWordList(accessToken, approvedWords)
addApprovedWordList(accessToken, approvedWords)
performApprovedWordsCheck(messageToCheck, delimiters) : WordList
approvedWordsAdded(wordList)
approvedWordsRemoved(wordList)
refreshApprovedWordListCache()
GUIDictionaryNavGroup
addDictionary(dictionary)
DictionaryWrapper
DataModel
ModelObserver
GUIModelObserver
javax.swing.JFrame
GUI
DictionaryPropertiesDialog
CosEvent.PushConsumer
InstallableModule
GUIDictionaryModule
get()

Figure 54. GUIDictionaryModuleClasses (Class Diagram)

3.7.1.1.1 CosEvent.PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.7.1.1.2 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.7.1.1.3 DictionaryPropertiesDialog (Class)

This dialog is the editing interface which allows the user to view, add, and remove banned words from a given dictionary.

3.7.1.1.4 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.7.1.1.5 GUIDictionaryNavGroup (Class)

This class is used to support the required Navigator functionality to group anu dictionary objects together for the purpose of being displayed together under one branch of the Navigator tree.

3.7.1.1.6 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

3.7.1.1.7 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

3.7.1.1.8 GUIDictionary (Class)

This class is a GUI wrapper for the Dictionary class. It adds functionality for caching the data and for adding GUI-specific functionality such as menus and Navigator support.

3.7.1.1.9 GUIDictionaryModule (Class)

This class is an installable GUI module that handles all of the dictionary-specific functionality in the GUI.

3.7.1.1.10 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.7.1.1.11 Navigable (Class)

This interface will be implemented by any class that supports the Navigator on either the left or right side (the tree or list view). This includes the functionality common to both the tree and list.

3.7.1.1.12 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.7.1.1.13 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

3.7.1.1.14 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.7.1.1.15 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerance by automatically failing over to a “working” dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.7.1.1.16 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.7.1.1.17 javax.swing.JFrame (Class)

Java class that displays a frame window.

3.7.1.1.18 ModelObserver (Class)

This interface must be implemented by any object that may need to be attached to the DataModel as an observer and get updated as system objects are added, deleted or changed.

3.7.2 Sequence Diagrams

3.7.2.1 GUIDictionaryModule:DictionaryApprovedWordProperties (Sequence Diagram)

This diagram shows how the editing of the approved words in a given dictionary will be done. It begins with a user clicking on a menu item from the GUIDictionary's context menu. Since the GUIDictionary will be an ActionListener for the menu item, the GUIDictionary will be called and then creates the DictionaryDialog. This dialog attaches itself as an observer to the DataModel in order to catch any updates to the word list (which will come through the event channel and then through the DataModel). It gets the list of banned and approved words and displays them to the user. This dialog shows both the banned word and approved word lists on two tabs. Approved word list can be viewed by selecting the Approved Words tab. When the user provides a list of approved words to add or remove, the GUIDictionary will make a call to the served Dictionary Wrapper object. If the words are added or removed successfully, the Dictionary object will push an event through the Dictionary event channel. (See the EventHandling diagram for details.) The DataModel will then call the dialog's update() method, and the dialog will ask the GUIDictionary wrapper for the current list of words to display. Just before the dialog is closed, it will detach from the DataModel.

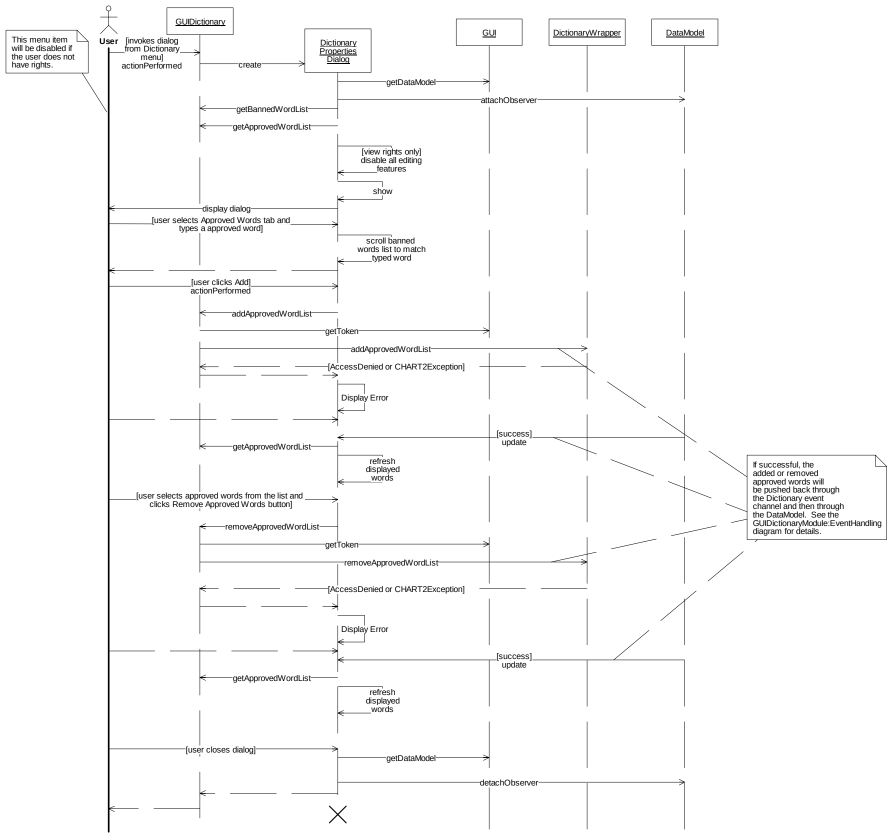

User
GUIDictionary
Dictionary Properties Dialog
GUI
DictionaryWrapper
DataModel
This menu item will be disabled if the user does not have rights.
[invokes dialog from Dictionary menu] actionPerformed
create
getDataModel
attachObserver
getBannedWordList
getApprovedWordList
[view rights only] disable all editing features
show
display dialog
[user selects Approved Words tab and types a approved word]
scroll banned words list to match typed word
[user clicks Add] actionPerformed
addApprovedWordList
getToken
addApprovedWordList
[AccessDenied or CHART2Exception]
Display Error
[success] update
getApprovedWordList
refresh displayed words
[user selects approved words from the list and clicks Remove Approved Words button]
removeApprovedWordList
getToken
removeApprovedWordList
[AccessDenied or CHART2Exception]
Display Error
[success] update
getApprovedWordList
refresh displayed words
[user closes dialog]
getDataModel
detachObserver
If successful, the added or removed approved words will be pushed back through the Dictionary event channel and then through the DataModel. See the GUIDictionaryModule:EventHandling diagram for details.

Figure 55. GUIDictionaryModule: DictionaryApprovedWordProperties (Sequence Diagram)

3.7.2.2 GUIDictionaryModule:DictionaryBannedWordProperties (Sequence Diagram)

This diagram shows how the editing of the banned words in a given dictionary will be done. It begins with a user clicking on a menu item from the GUIDictionary's context menu. Since the GUIDictionary will be an ActionListener for the menu item, the GUIDictionary will be called and then creates the DictionaryDialog. This dialog attaches itself as an observer to the DataModel in order to catch any updates to the word list (which will come through the event channel and then through the DataModel). It gets the list of banned and approved words and displays them to the user. This dialog shows both the banned word and approved word lists on two tabs. Banned word list is displayed on the top. When the user provides a list of banned words to add or remove, the GUIDictionary will make a call to the served Dictionary Wrapper object. If the words are added or removed successfully, the Dictionary object will push an event through the Dictionary event channel. (See the EventHandling diagram for details.) The DataModel will then call the dialog's update() method, and the dialog will ask the GUIDictionary wrapper for the current list of words to display. Just before the dialog is closed, it will detach from the DataModel.

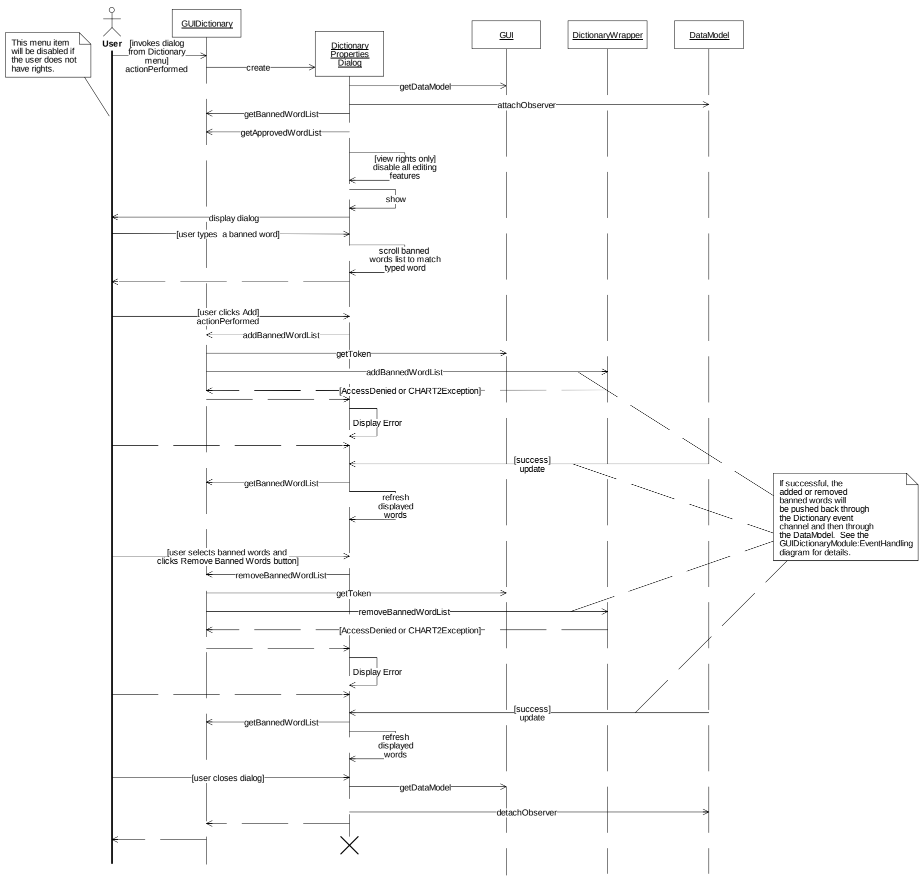

User
GUIDictionary
Dictionary Properties Dialog
GUI
DictionaryWrapper
DataModel
This menu item will be disabled if the user does not have rights.
[invokes dialog from Dictionary menu] actionPerformed
create
getDataModel
attachObserver
getBannedWordList
getApprovedWordList
[view rights only] disable all editing features
show
display dialog
[user types a banned word]
scroll banned words list to match typed word
[user clicks Add] actionPerformed
addBannedWordList
getToken
addBannedWordList
[AccessDenied or CHART2Exception]
Display Error
[success] update
getBannedWordList
refresh displayed words
[user selects banned words and clicks Remove Banned Words button]
removeBannedWordList
getToken
removeBannedWordList
[AccessDenied or CHART2Exception]
Display Error
[success] update
getBannedWordList
refresh displayed words
[user closes dialog]
getDataModel
detachObserver
If successful, the added or removed banned words will be pushed back through the Dictionary event channel and then through the DataModel. See the GUIDictionaryModule:EventHandling diagram for details.

Figure 56. GUIDictionaryModule:DictionaryBannedWordProperties (Sequence Diagram)

3.7.2.3 GUIDictionaryModule:Discovery (Sequence Diagram)

This diagram shows how the Dictionary event channels and Dictionary objects are discovered and added to the system. This will be a periodic process, and the GUI will call the GUIDictionaryModule repeatedly. When the GUI asks the module to discover event channels, it looks up the Dictionary event channels in the trader. It then creates a PushEventConsumer and adds it to the EventConsumerGroup, which actually attaches the consumer to the channel and reattaches it if the event service is restarted. (Duplicate channels will be ignored). The GUI then calls the module to discover objects. At this time the module will query the Dictionary Wrapper objects in the trader. If any are found, it will create an Identifier to be used as a lookup key for use with the DataModel. If the GUIDictionary wrapper object does not already exist in the DataModel, it is created and added. Creating the GUI wrapper will cause the new wrapper to initialize its data by making a remote call to the served Dictionary Wrapper object. The GUIDictionary is then added to the GUIDictionaryNavGroup and the DataModel is called to propagate the changes to any interested observers such as the DictionaryPropertiesDialog.

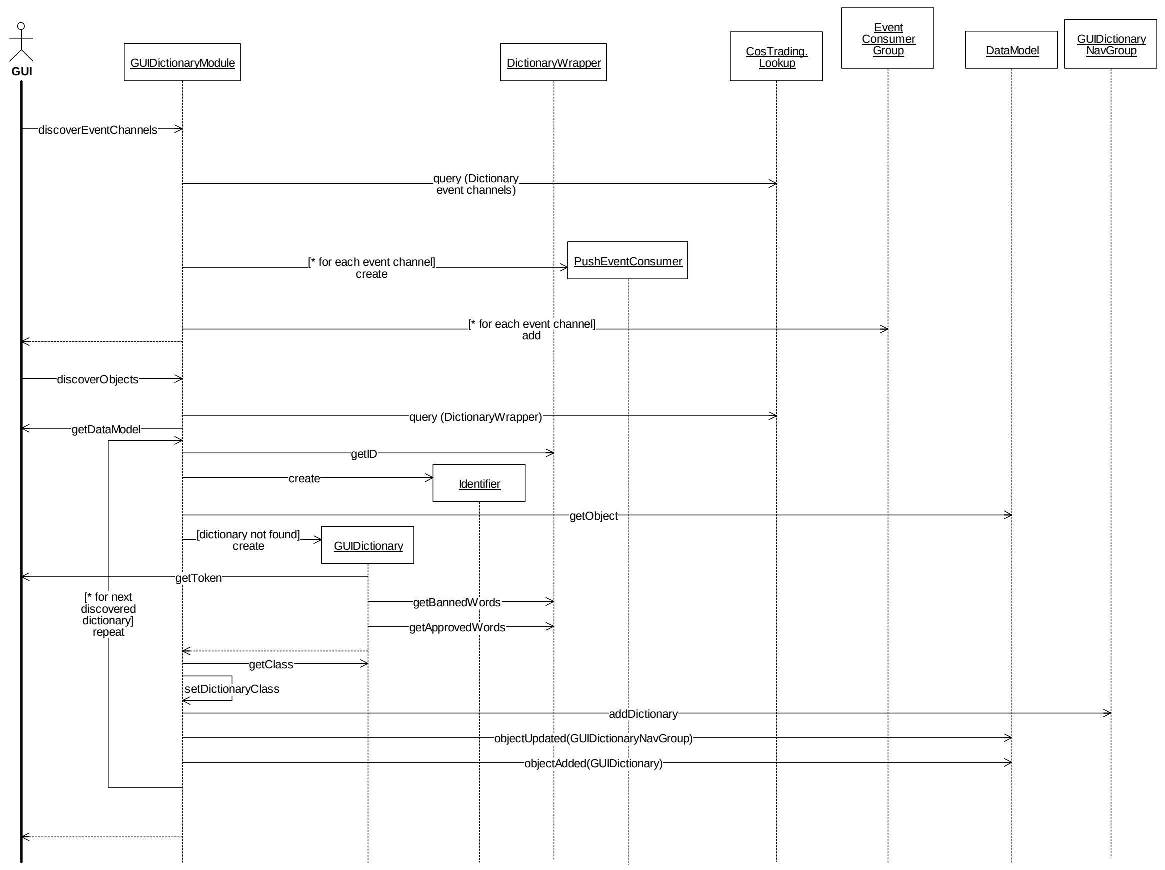
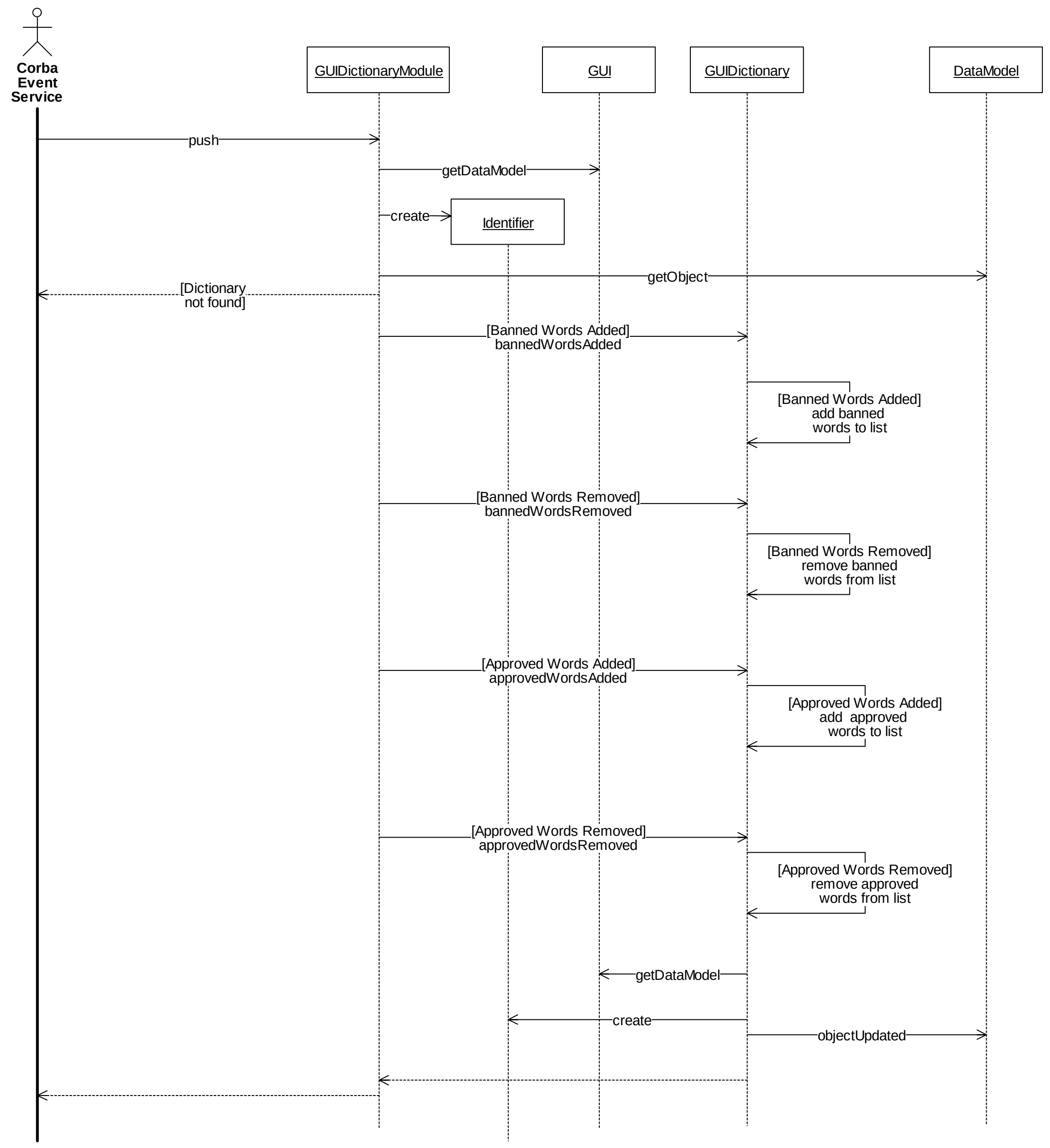


Figure 57. GUIDictionaryModule:Discovery (Sequence Diagram)

3.7.2.4 GUIDictionaryModule:EventHandling (Sequence Diagram)

This diagram shows how dictionary events are propagated through the GUI when they are pushed from the event channel. The ORB invokes the push method of the DictionaryEventConsumer. The event data contains a byte array identifier, which is used to create an Identifier object to get the GUIDictionary object from the DataModel. The words are added or removed from the wrapper's cache, and then the DataModel is called to update any observers that may be listening for updates, such as the DictionaryPropertiesDialog.



Corba Event Service
GUIDictionaryModule
GUI
GUIDictionary
DataModel
push
getDataModel
create
Identifier
getObject
[Dictionary not found]
[Banned Words Added] bannedWordsAdded
[Banned Words Added] add banned words to list
[Banned Words Removed] bannedWordsRemoved
[Banned Words Removed] remove banned words from list
[Approved Words Added] approvedWordsAdded
[Approved Words Added] add approved words to list
[Approved Words Removed] approvedWordsRemoved
[Approved Words Removed] remove approved words from list
getDataModel
create
objectUpdated

Figure 58. GUIDictionaryModule:EventHandling (Sequence Diagram)

3.7.2.5 GUIDictionaryModule:Shutdown (Sequence Diagram)

This diagram shows what happens at shutdown. The module deactivated from the POA to clean up.

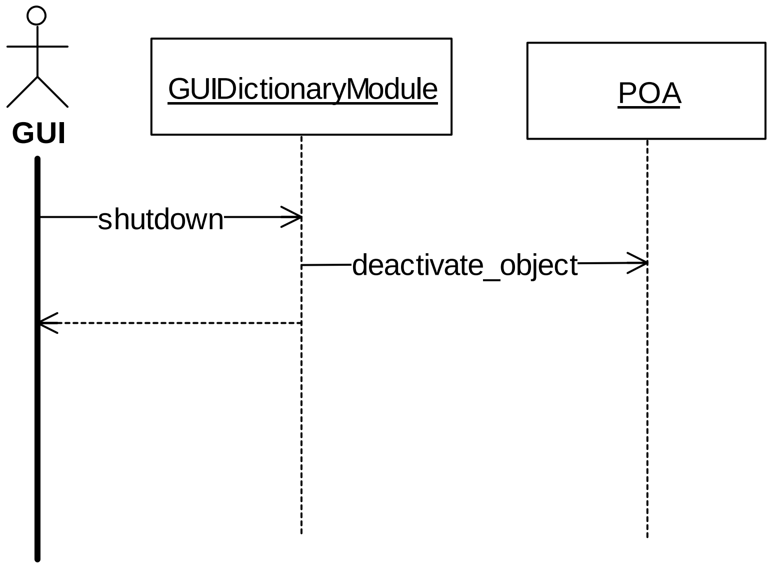


Figure 59. GUIDictionaryModule:Shutdown (Sequence Diagram)

3.7.2.6 GUIDictionaryModule:Startup (Sequence Diagram)

This diagram show the steps taken to initialize the GUIDictionaryModule. The GUI will call the module's startup method. The module will create a GUIDictionaryNavGroup and add it to the DataModel so that the Navigator will display it. The module will store the group for later use. The GUIDictionaryModule is activated using the POA so that it can serve as a PushConsumer to receive dictionary events.

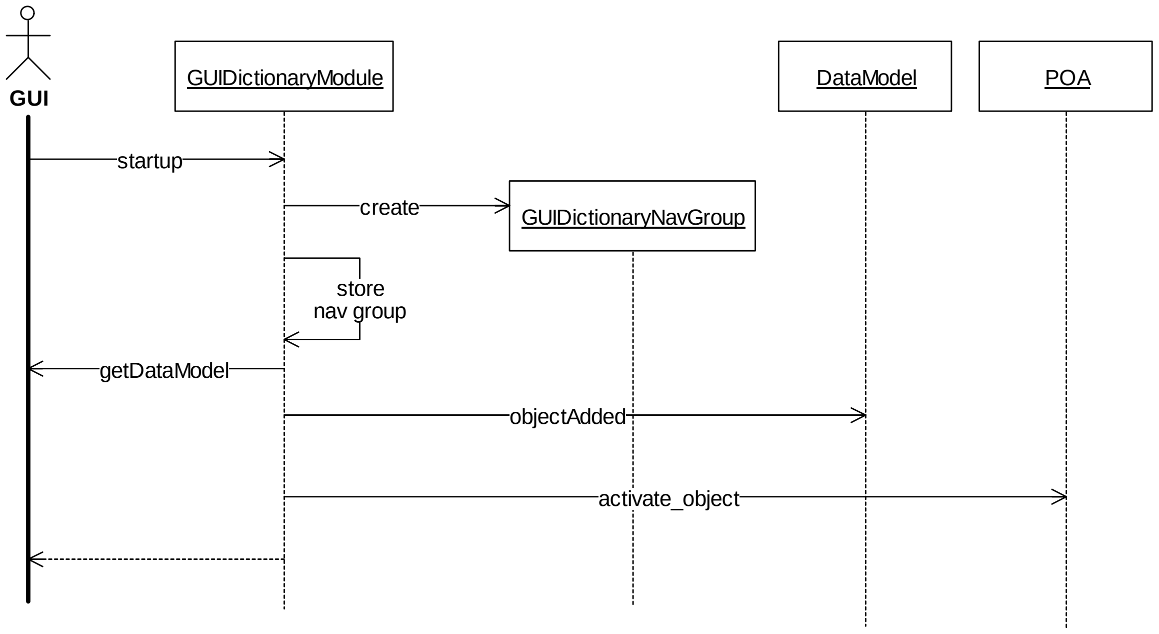


Figure 60. GUIDictionaryModule:Startup (Sequence Diagram)

3.8 GUIHARModule

3.8.1 Class Diagrams

3.8.1.1 Dialogs (Class Diagram)

This diagram shows all of the classes representing windows that exist within the GUIHARModule.

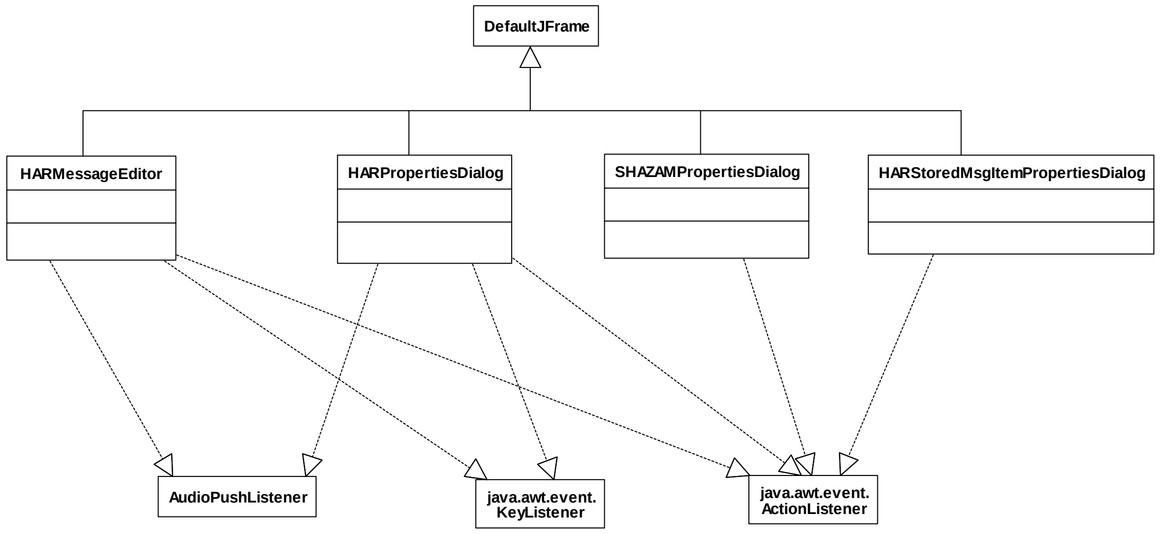


Figure 61. Dialogs (Class Diagram)

3.8.1.1.1 AudioPushListener (Class)

This is called by one or more AudioPushConsumerImpls when an audio clip is being pushed.

3.8.1.1.2 DefaultJFrame (Class)

This class provides a default implementation of the WindowManageable interface, and may be used as a base class for other frame windows in the GUI. It handles all interactions with the WindowManager for attaching and detaching, as well as saving the window position.

3.8.1.1.3 HARMessageEditor (Class)

This dialog is used for creating a new HAR stored message, viewing or modifying an existing HAR stored message, and setting the message while the HAR is in maintenance mode.

3.8.1.1.4 HARPropertiesDialog (Class)

This dialog is used to view and edit the HAR’s configuration, and to view and edit the current slot contents.

3.8.1.1.5 HARStoredMsgItemPropertiesDialog (Class)

This dialog is used for creating, viewing, or editing the properties of HARStoredMsgItem / GUIHARStoredMsgItem objects.

3.8.1.1.6 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.8.1.1.7 java.awt.event. KeyListener (Class)

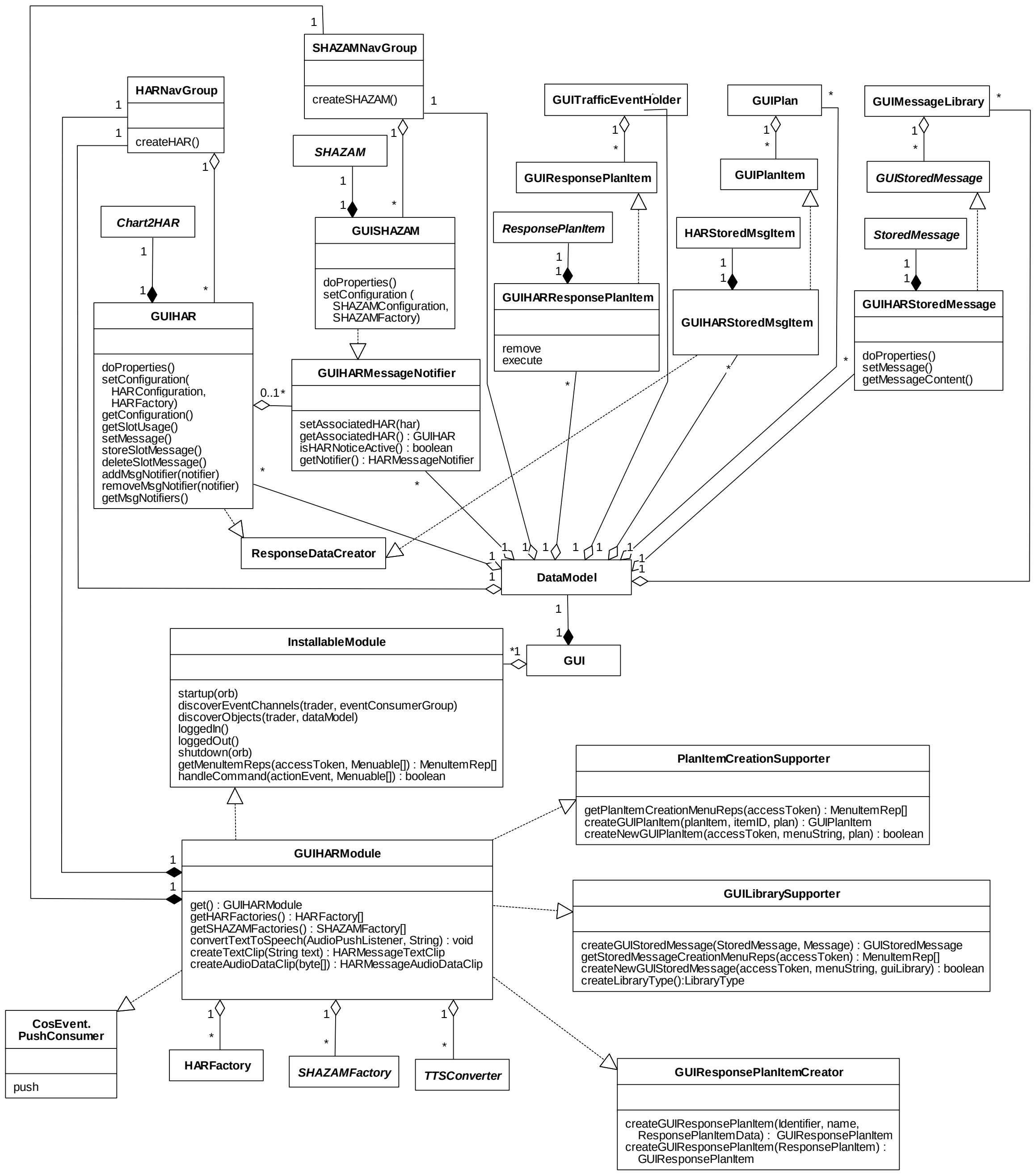
Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

3.8.1.1.8 SHAZAMPropertiesDialog (Class)

This dialog is used for viewing and editing the properties (configuration) of a SHAZAM.

3.8.1.2 HARModuleArchitecture (Class Diagram)

This diagram shows the data hierarchy of the GUIHARModule and the objects it supports. It does not contain the user interface relationships of these objects - those are contained in the GUIHARModule:NavigatorSupport class diagram.



SHAZAMNavGroup
createSHAZAM()
HARNavGroup
createHAR()
SHAZAM
Chart2HAR
GUISHAZAM
doProperties()
setConfiguration (SHAZAMConfiguration, SHAZAMFactory)
GUIHAR
doProperties()
setConfiguration(HARConfiguration, HARFactory)
getConfiguration()
getSlotUsage()
setMessage()
storeSlotMessage()
deleteSlotMessage()
addMsgNotifier(notifier)
removeMsgNotifier(notifier)
getMsgNotifiers()
GUIHARMessageNotifier
setAssociatedHAR(har)
getAssociatedHAR() : GUIHAR
isHARNoticeActive() : boolean
getNotifier() : HARMessageNotifier
GUITrafficEventHolder
GUIResponsePlanItem
ResponsePlanItem
GUIHARResponsePlanItem
remove
execute
GUIPlan
GUIPlanItem
HARStoredMsgItem
GUIHARStoredMsgItem
GUIMessageLibrary
GUIStoredMessage
StoredMessage
GUIHARStoredMessage
doProperties()
setMessage()
getMessageContent()
ResponseDataCreator
DataModel
InstallableModule
startup(orb)
discoverEventChannels(trader, eventConsumerGroup)
discoverObjects(trader, dataModel)
loggedIn()
loggedOut()
shutdown(orb)
getMenuItemReps(accessToken, Menuable[]) : MenuItemRep[]
handleCommand(actionEvent, Menuable[]) : boolean
GUI
PlanItemCreationSupporter
getPlanItemCreationMenuReps(accessToken) : MenuItemRep[]
createGUIPlanItem(planItem, itemID, plan) : GUIPlanItem
createNewGUIPlanItem(accessToken, menuString, plan) : boolean
GUIHARModule
get() : GUIHARModule
getHARFactories() : HARFactory[]
getSHAZAMFactories() : SHAZAMFactory[]
convertTextToSpeech(AudioPushListener, String) : void
createTextClip(String text) : HARMessageTextClip
createAudioDataClip(byte[]) : HARMessageAudioDataClip
GUILibrarySupporter
createGUIStoredMessage(StoredMessage, Message) : GUIStoredMessage
getStoredMessageCreationMenuReps(accessToken) : MenuItemRep[]
createNewGUIStoredMessage(accessToken, menuString, guiLibrary) : boolean
createLibraryType():LibraryType
CosEvent. PushConsumer
push
HARFactory
SHAZAMFactory
TTSConverter
GUIResponsePlanItemCreator
createGUIResponsePlanItem(Identifier, name, ResponsePlanItemData) : GUIResponsePlanItem
createGUIResponsePlanItem(ResponsePlanItem) : GUIResponsePlanItem
0..1*
*1

Figure 62. HARModuleArchitecture (Class Diagram)

3.8.1.2.1 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.8.1.2.2 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.8.1.2.3 GUIHAR (Class)

This class provides a GUI "wrapper" object that is used to wrap the CHART2HAR CORBA interface and to supply GUI-specific functionality.

3.8.1.2.4 GUIHARModule (Class)

The GUIHARModule is an installable module in the GUI, and provides all functionality specific to HAR and SHAZAM control. It requires that the GUIPlanModule, the GUILibraryModule, and the GUITrafficEventModule all be installed in order to be fully functional. If any of the other modules is not available, the functionality provided by that module will not be available. For example, if the GUILibraryModule is not installed, the user will not be able to create or utilize HAR library messages. Only one GUIHARModule object may exist within the GUI. This class implements the interfaces to support the frameworks of the GUIPlanModule, the GUILibraryModule, and the GUITrafficEventModule.

3.8.1.2.5 GUIHARResponsePlanItem (Class)

This class provides a GUI "wrapper" object that is used to wrap a ResponsePlanItem CORBA interface that contains HAR-specific data and to supply GUI-specific functionality.

3.8.1.2.6 GUIHARStoredMessage (Class)

This class provides a GUI "wrapper" object that is used to wrap a StoredMessage CORBA interface that contains HAR-specific data and to supply GUI-specific functionality.

3.8.1.2.7 GUIHARStoredMsgItem (Class)

This class provides a GUI “wrapper” object that is used to wrap the HARStoredMsgItem CORBA interface and to supply GUI-specific functionality.

3.8.1.2.8 GUIPlan (Class)

This class is a GUI wrapper for the Plan object. The wrapping is done to cache the data locally for faster access, as well as to give the Plan some GUI-specific functionality such as menus and command handling.

3.8.1.2.9 GUIPlanItem (Class)

This is a GUI base class for all the plan items. Each GIUPlanItem object will serve as a GUI wrapper to cache the plan item data locally and also to handle all user interaction in the GUI, such as menus and command handling.

3.8.1.2.10 GUITrafficEventHolder (Class)

This object represents a TrafficEvent and provides GUI functionality for the TrafficEvent. This class contains generic data and operations that apply to any type of TrafficEvent. It also “holds” a type-specific GUITrafficEvent. If the type of the TrafficEvent is changed, the old GUITrafficEvent object (stored within this “holder” class) will be switched out for a new GUITrafficEvent of a different type, but the GUITrafficEventHolder will remain in existence.

3.8.1.2.11 GUIResponsePlanItem (Class)

This is a base class for the GUI wrapper object that is used to wrap a ResponsePlanItem. The ResponsePlanItem represents a proposed action to perform on a target object in response to a TrafficEvent. This wrapper object adds GUI-specific functionality to the response plan item.

3.8.1.2.12 GUIStoredMessage (Class)

This class is a GUI “wrapper” object that is used to wrap a StoredMessage object. It provides a user interface object which can implement whatever interfaces are necessary for the object to exist within the GUI framework (for example, an object must support the NavTreeDisplayable and/or NavListDisplayable interface to be displayed in the Navigator).

3.8.1.2.13 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description, which are used to allow the user to organize messages.

3.8.1.2.14 PlanItemCreationSupporter (Class)

This interface must be implemented in any modules that wish to support the plan module. The modules must attach their PlanItemCreationSupporters at startup. The GUIPlanModule will then call the supporter when it is time to display the Plan menu or to create a specific type of plan item or GUIPlanItem.

3.8.1.2.15 GUIHARMessageNotifier (Class)

This interface is similar to the HARMessageNotifier interface in that it is implemented by all of the message notifier classes, but this interface is specific to the GUIHARModule and its usage of the GUI wrapper objects.

3.8.1.2.16 ResponseDataCreator (Class)

This interface enables the creation of type-specific ResponsePlanItemData objects, which are used for creating the appropriate type of ResponsePlanItem. An object implementing this interface can be added to the response plan of a traffic event. Implementers of this interface include plan items and response devices.

3.8.1.2.17 Chart2HAR (Class)

The Chart2HAR class is an extension of the HAR that is aware of Chart2 business rules, such as arbitration queues, linking device usage to traffic events, and the concept of a shared resource.

3.8.1.2.18 GUIMessageLibrary (Class)

This class is a GUI “wrapper” object that is used to wrap a MessageLibrary object. The wrapping is done to cache the data locally for faster access, as well as to give the MessageLibrary some GUI-specific functionality such as menus and command handling.

3.8.1.2.19 GUISHAZAM (Class)

This class is a GUI wrapper object that is used to wrap a SHAZAM CORBA interface and to provide GUI-specific functionality.

3.8.1.2.20 HARFactory (Class)

This CORBA interface allows new HAR objects to be added to the system.

3.8.1.2.21 SHAZAM (Class)

This class is used to represent a SHAZAM field device. This class uses a helper class to perform the model specific protocol for device command and control.

3.8.1.2.22 GUILibrarySupporter (Class)

This class allows the GUILibraryModule to maintain stored messages that have differing formats. When an object of this type is installed the user can create, maintain, and use the specific type of libraries and stored messages that the object supports.

3.8.1.2.23 HARNavGroup (Class)

This class has one instance in the GUIHARModule. It serves as a container for all of the GUIHAR objects in the module when they are displayed in the Navigator.

3.8.1.2.24 HARStoredMsgItem (Class)

This class provides a means for associating a HAR message with a HAR for use in responding to a traffic event. A directional indicator is stored to specify the SHAZAMs to activate (by default) when the message is activated on the specified HAR.

3.8.1.2.25 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.8.1.2.26 ResponsePlanItem (Class)

Objects of this type can be executed as part of a traffic event response plan. A ResponsePlanItem can be executed by an operator, at which time it becomes the responsibility of the System to activate the item on the ResponseDevice as soon as it is appropriate.

3.8.1.2.27 TTSConverter (Class)

This interface represents the Text to Speech converter object that allows text to be passed in and speech to be returned.

3.8.1.2.28 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.8.1.2.29 GUIResponsePlanItemCreator (Class)

This interface is used to enable the creation of specific types of GUIResponsePlanItem wrapper objects depending upon which type of ResponsePlanItem is being wrapped. Any class wishing to create GUIResponsePlanItems must implement this interface and add themselves to the GUITrafficEventModule at GUI startup time. When the GUITrafficEventModule discovers a ResponsePlanItem or catches a CORBA event indicating that a new response plan item has been created, it will call each known GUIResponsePlanItemCreator to give it an opportunity to create a specific type of GUI wrapper object.

3.8.1.2.30 SHAZAMFactory (Class)

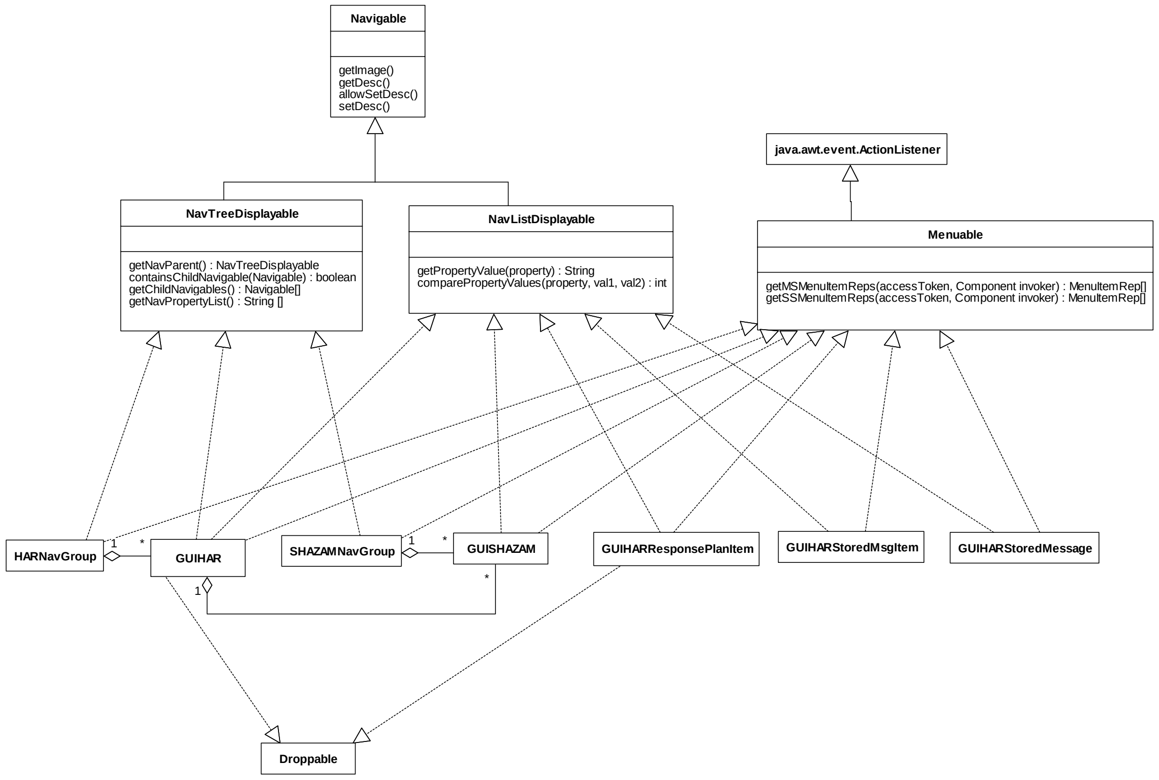
This CORBA interface allows new SHAZAM objects to be added to the system.

3.8.1.2.31 SHAZAMNavGroup (Class)

This class has one instance in the GUIHARModule. It serves as a container for all of the GUISHAZAM objects in the module when they are displayed in the Navigator.

3.8.1.3 NavigatorSupport (Class Diagram)

This diagram shows the user interface relationships of the objects supported by the GUIHARModule.



Navigable
getImage()
getDesc()
allowSetDesc()
setDesc()
java.awt.event.ActionListener
NavTreeDisplayable
getNavParent() : NavTreeDisplayable
containsChildNavigable(Navigable) : boolean
getChildNavigables() : Navigable[]
getNavPropertyList() : String []
NavListDisplayable
getPropertyValue(property) : String
comparePropertyValues(property, val1, val2) : int
Menuable
getMSMenuItemReps(accessToken, Component invoker) : MenuItemRep[]
getSSMenuItemReps(accessToken, Component invoker) : MenuItemRep[]
HARNavGroup
GUIHAR
SHAZAMNavGroup
GUISHAZAM
GUIHARResponsePlanItem
GUIHARStoredMsgItem
GUIHARStoredMessage
Droppable
1
*
1
*
1
*

Figure 63. NavigatorSupport (Class Diagram)

3.8.1.3.1 GUIHAR (Class)

This class provides a GUI “wrapper” object that is used to wrap the CHART2HAR CORBA interface and to supply GUI-specific functionality.

3.8.1.3.2 GUIHARResponsePlanItem (Class)

This class provides a GUI “wrapper” object that is used to wrap a ResponsePlanItem CORBA interface that contains HAR-specific data and to supply GUI-specific functionality.

3.8.1.3.3 Droppable (Class)

This interface must be implemented by any object wishing to take part in a drag and drop operation. It is used by the DropHandler class to determine if a drop action should be allowed and to delegate the handling of the drop action after it is performed.

3.8.1.3.4 GUIHARStoredMessage (Class)

This class provides a GUI “wrapper” object that is used to wrap a StoredMessage CORBA interface that contains HAR-specific data and to supply GUI-specific functionality.

3.8.1.3.5 GUIHARStoredMsgItem (Class)

This class provides a GUI “wrapper” object that is used to wrap the HARStoredMsgItem CORBA interface and to supply GUI-specific functionality.

3.8.1.3.6 GUISHAZAM (Class)

This class is a GUI wrapper object that is used to wrap a SHAZAM CORBA interface and to provide GUI-specific functionality.

3.8.1.3.7 HARNavGroup (Class)

This class has one instance in the GUIHARModule. It serves as a container for all of the GUIHAR objects in the module when they are displayed in the Navigator.

3.8.1.3.8 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user’s access rights before supplying the strings, so the user’s actions may be restricted.

3.8.1.3.9 Navigable (Class)

This interface will be implemented by any class that supports the Navigator on either the left or right side (the tree or list view). This includes the functionality common to both the tree and list.

3.8.1.3.10 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.8.1.3.11 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

3.8.1.3.12 SHAZAMNavGroup (Class)

This class has one instance in the GUIHARModule. It serves as a container for all of the GUISHAZAM objects in the module when they are displayed in the Navigator.

3.8.1.3.13 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.8.2 Sequence Diagrams

3.8.2.1 GUIHARModule:AddHAR (Sequence Diagram)

This diagram shows how a HAR is added to the system. The user right clicks on the HARNavGroup in the Navigator and clicks “Add HAR”. The HARNavGroup then creates a temporary GUIHAR wrapper object and calls it to display its properties, which invokes the HAR Properties dialog. When the user clicks OK, the dialog calls the GUIHAR to set the configuration. The GUIHAR wrapper object does not contain a served CHART2HAR object, so it calls the HARFactory to create one. If a new CHART2HAR is successfully created, the server will push out an event and the GUI will create a new GUIHAR object to wrap it. The temporary GUIHAR object will be deleted.

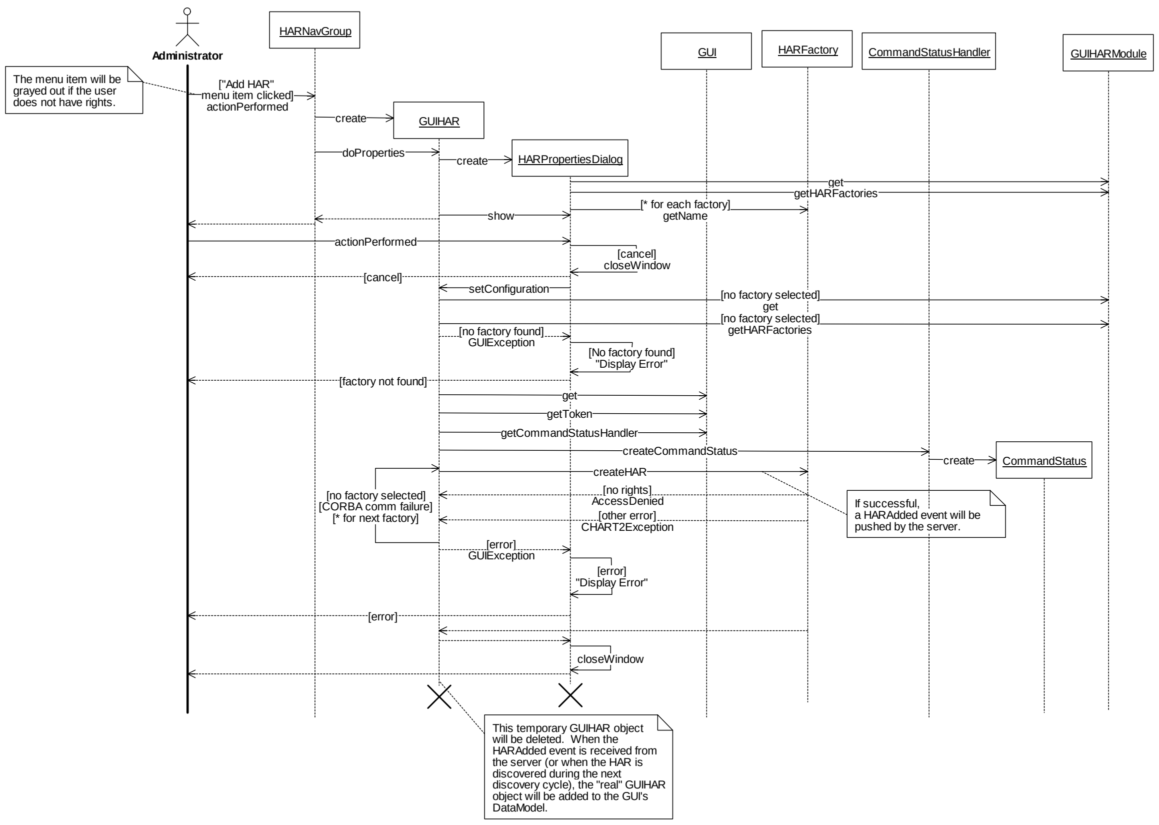

Administrator
HARNavGroup
GUI
HARFactory
CommandStatusHandler
GUIHARModule
The menu item will be grayed out if the user does not have rights.
["Add HAR" menu item clicked] actionPerformed
create
GUIHAR
doProperties
create
HARPropertiesDialog
get
getHARFactories
[* for each factory] getName
show
actionPerformed
[cancel] closeWindow
[cancel]
setConfiguration
[no factory selected] get
[no factory selected] getHARFactories
[no factory found] GUIException
[No factory found] "Display Error"
[factory not found]
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
createHAR
[no factory selected] [CORBA comm failure] [* for next factory]
[no rights] AccessDenied
[other error] CHART2Exception
If successful, a HARAdded event will be pushed by the server.
[error] GUIException
[error] "Display Error"
[error]
closeWindow
This temporary GUIHAR object will be deleted. When the HARAdded event is received from the server (or when the HAR is discovered during the next discovery cycle), the "real" GUIHAR object will be added to the GUI's DataModel.

Figure 64. GUIHARModule:AddHAR (Sequence Diagram)

3.8.2.2 GUIHARModule:AddHARStoredMessageItem (Sequence Diagram)

This diagram shows how a PlanItem is added to the system. The user clicks on the GUIPlan object in the Navigator and chooses “Create HAR Plan Item”. The GUIPlan then calls the PlanItemCreationSupporters (of which the GUIHARModule is one) to create the GUIPlanItem, and the GUIHARModule recognizes the menu item string. The module creates a temporary GUI wrapper for a plan item and calls it to display its properties, which invokes the HARStoredMsgItemPropertiesDialog. When the user clicks Apply or OK, the dialog calls back to the GUIHARStoredMsgItem wrapper object to set the item data. Since the wrapper contains no served PlanItem, it calls the CHART2HAR to create one. If successful, the server will push a PlanItemAdded event to all GUIs, which the GUI will catch to create a new GUIHARStoredMsgItem wrapper object (the temporary wrapper will be deleted).

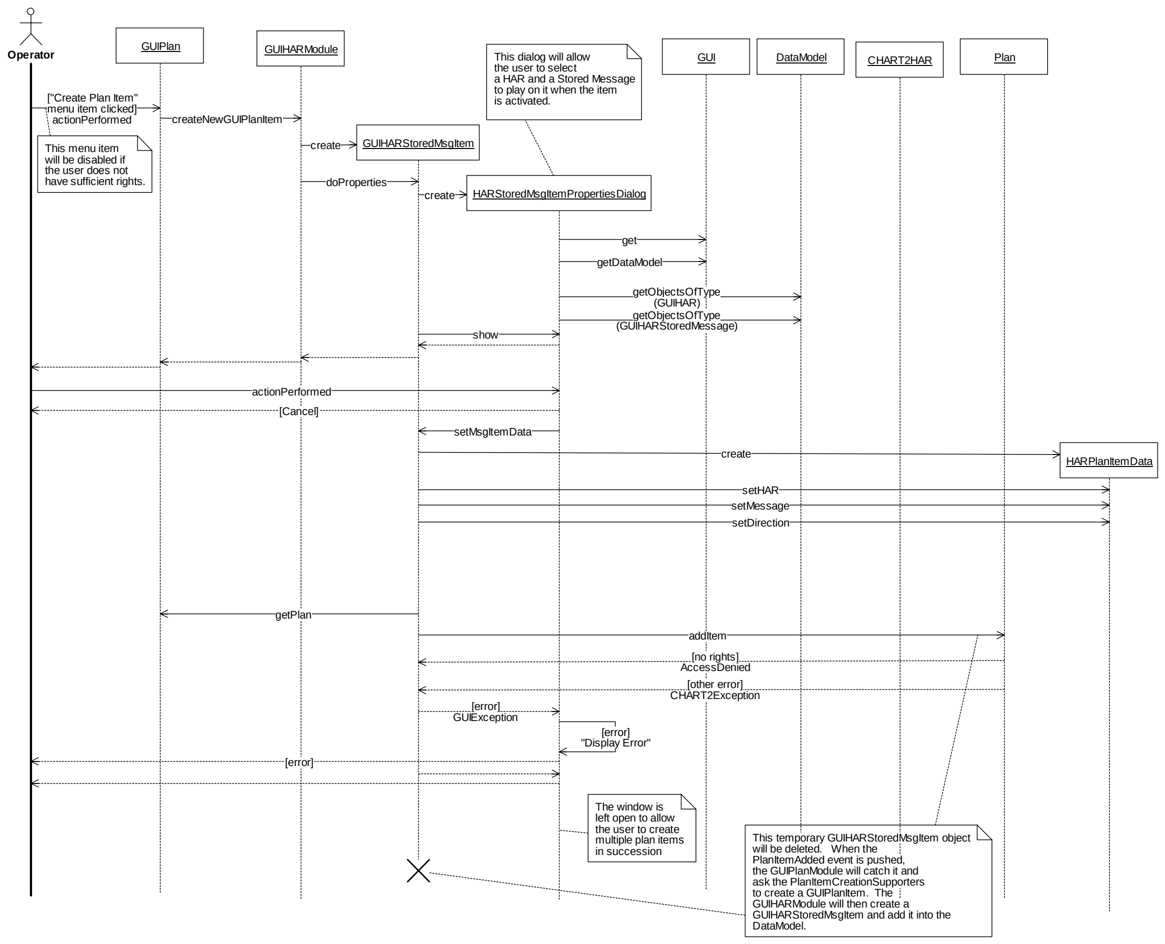
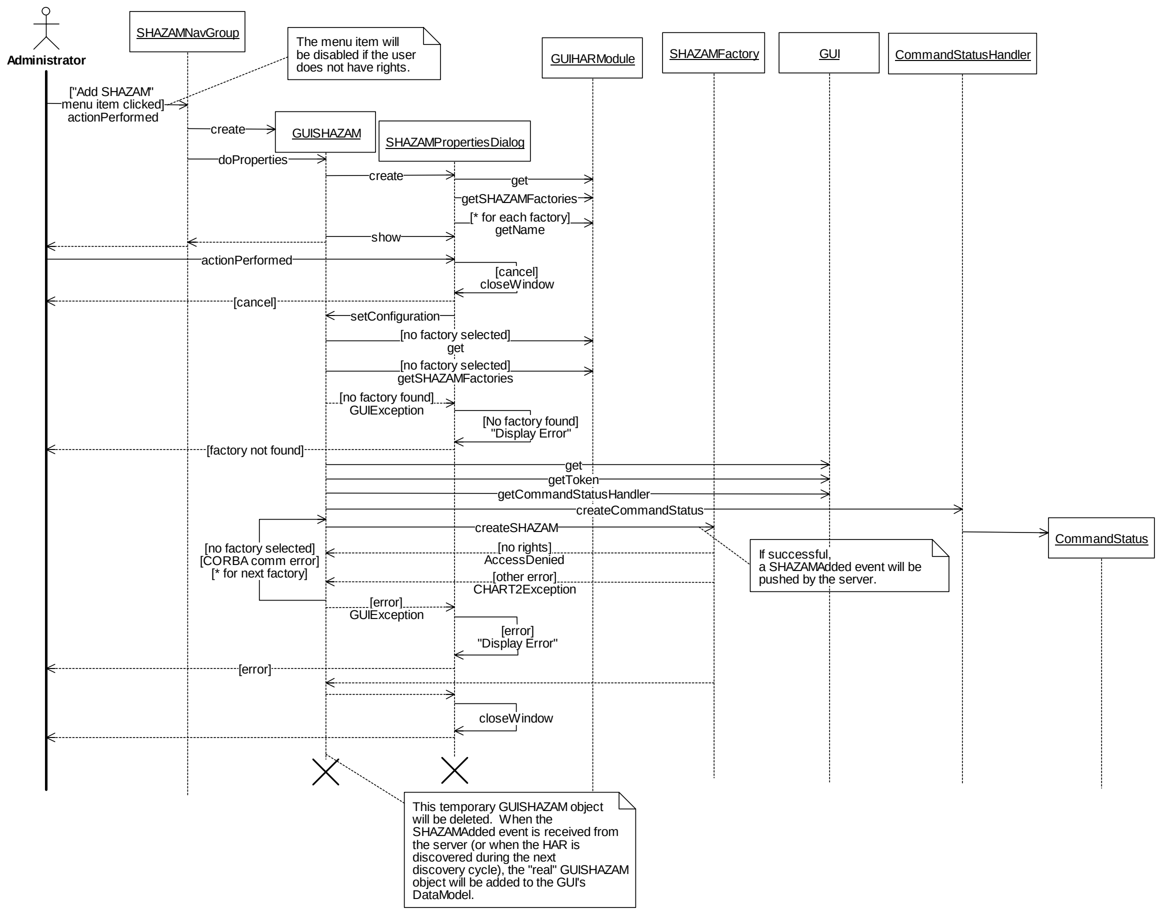

Operator
GUIPlan
GUIHARModule
This dialog will allow the user to select a HAR and a Stored Message to play on it when the item is activated.
GUI
DataModel
CHART2HAR
Plan
["Create Plan Item" menu item clicked] actionPerformed
createNewGUIPlanItem
This menu item will be disabled if the user does not have sufficient rights.
create
GUIHARStoredMsgItem
doProperties
create
HARStoredMsgItemPropertiesDialog
get
getDataModel
getObjectsOfType (GUIHAR)
getObjectsOfType (GUIHARStoredMessage)
show
actionPerformed
[Cancel]
setMsgItemData
create
HARPlanItemData
setHAR
setMessage
setDirection
getPlan
addItem
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
The window is left open to allow the user to create multiple plan items in succession
This temporary GUIHARStoredMsgItem object will be deleted. When the PlanItemAdded event is pushed, the GUIPlanModule will catch it and ask the PlanItemCreationSupporters to create a GUIPlanItem. The GUIHARModule will then create a GUIHARStoredMsgItem and add it into the DataModel.

Figure 65. GUIHARModule:AddHARStoredMessageItem (Sequence Diagram)

3.8.2.3 GUIHARModule:AddSHAZAM (Sequence Diagram)

This diagram shows how a SHAZAM is added to the system. The user right clicks on the SHAZAMNavGroup object in the Navigator and clicks “Add SHAZAM”. The SHAZAMNavGroup then creates a temporary GUISHAZAM wrapper object and calls it to display its properties, which invokes the SHAZAM Properties dialog. When the user clicks OK, the dialog calls the GUISHAZAM to set the configuration. The GUISHAZAM wrapper object does not contain a served SHAZAM object, so it calls the SHAZAMFactory to create one. If a new SHAZAM is successfully created, the server will push out an event and the GUI will create a new GUISHAZAM object to wrap it. The temporary GUISHAZAM object will be deleted.



Administrator
SHAZAMNavGroup
The menu item will be disabled if the user does not have rights.
GUIHARModule
SHAZAMFactory
GUI
CommandStatusHandler
["Add SHAZAM" menu item clicked] actionPerformed
create
GUISHAZAM
SHAZAMPropertiesDialog
doProperties
create
get
getSHAZAMFactories
[* for each factory] getName
show
actionPerformed
[cancel] closeWindow
[cancel]
setConfiguration
[no factory selected] get
[no factory selected] getSHAZAMFactories
[no factory found] GUIException
[No factory found] "Display Error"
[factory not found]
get
getToken
getCommandStatusHandler
createCommandStatus
CommandStatus
createSHAZAM
[no factory selected] [CORBA comm error] [* for next factory]
[no rights] AccessDenied
[other error] CHART2Exception
If successful, a SHAZAMAdded event will be pushed by the server.
[error] GUIException
[error] "Display Error"
[error]
closeWindow
This temporary GUISHAZAM object will be deleted. When the SHAZAMAdded event is received from the server (or when the HAR is discovered during the next discovery cycle), the "real" GUISHAZAM object will be added to the GUI's DataModel.

Figure 66. GUIHARModule:AddSHAZAM (Sequence Diagram)

3.8.2.4 GUIHARModule:AssociateMessageNotifier (Sequence Diagram)

This diagram shows how a HAR message notifier (SHAZAM or DMS) is associated with a HAR. The administrator drags the GUIHARMessageNotifier object over the GUIHAR object in the Navigator. The drop will be rejected if the HARNotifier is active. When the object is dropped onto the GUIHAR, the GUIHAR wrapper calls the CHART2HAR server object to add the message notifier. If successful, the server will push an event and the GUI will catch the event and associate the GUIHAR wrapper object with the GUIHARMessageNotifier.

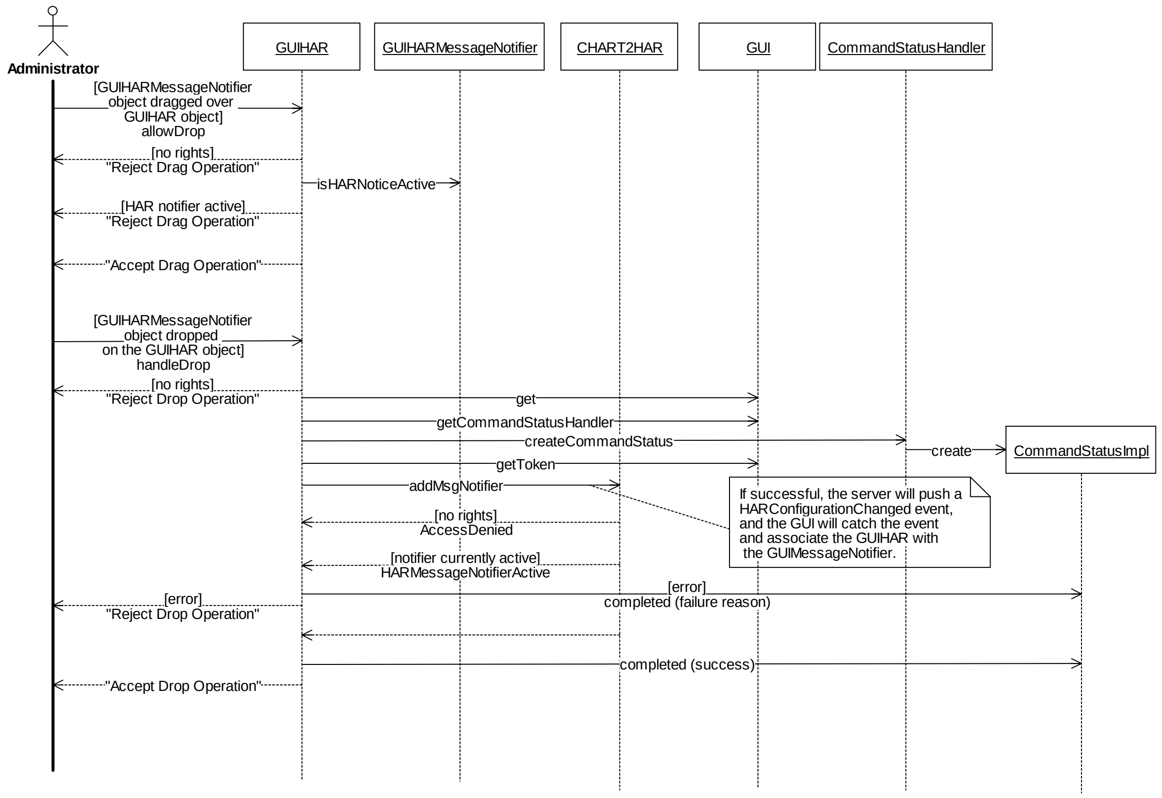

Administrator
GUIHAR
GUIHARMessageNotifier
CHART2HAR
GUI
CommandStatusHandler
[GUIHARMessageNotifier object dragged over GUIHAR object] allowDrop
[no rights] "Reject Drag Operation"
isHARNoticeActive
[HAR notifier active] "Reject Drag Operation"
"Accept Drag Operation"
[GUIHARMessageNotifier object dropped on the GUIHAR object] handleDrop
[no rights] "Reject Drop Operation"
get
getCommandStatusHandler
createCommandStatus
create
CommandStatusImpl
getToken
addMsgNotifier
If successful, the server will push a HARConfigurationChanged event, and the GUI will catch the event and associate the GUIHAR with the GUIMessageNotifier.
[no rights] AccessDenied
[notifier currently active] HARMessageNotifierActive
[error] completed (failure reason)
[error] "Reject Drop Operation"
completed (success)
"Accept Drop Operation"

Figure 67. GUIHARModule:AssociateMessageNotifier (Sequence Diagram)

3.8.2.5 GUIHARModule:BlankHAR (Sequence Diagram)

This diagram shows how a HAR is blanked when it is online. To blank the HAR, the response item must be removed from the event or the event must be closed. This may be done by right clicking on the GUIHARResponsePlanItem or on the GUITrafficEvent objects, respectively, and choosing the appropriate menu item. Either way, the remove method of the GUIHARResponsePlanItem wrapper object will be called, which will in turn call the served ResponsePlanItem object which it wraps. If successful, the server will push events to all GUIs indicating the changed status.

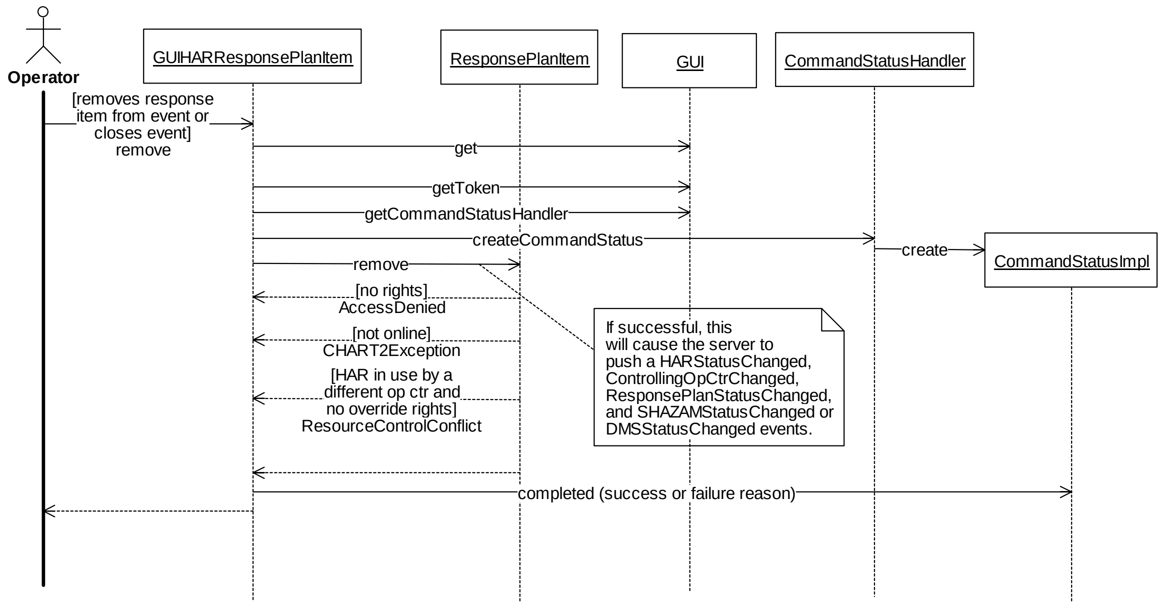

Operator
GUIHARResponsePlanItem
ResponsePlanItem
GUI
CommandStatusHandler
CommandStatusImpl
[removes response item from event or closes event] remove
get
getToken
getCommandStatusHandler
createCommandStatus
create
remove
[no rights] AccessDenied
[not online] CHART2Exception
[HAR in use by a different op ctr and no override rights] ResourceControlConflict
If successful, this will cause the server to push a HARStatusChanged, ControllingOpCtrChanged, ResponsePlanStatusChanged, and SHAZAMStatusChanged or DMSStatusChanged events.
completed (success or failure reason)

Figure 68. GUIHARModule:BlankHAR (Sequence Diagram)

3.8.2.6 GUIHARModule:BlankHARInMaintenanceMode (Sequence Diagram)

This diagram shows how a HAR is blanked when it is in maintenance mode. The user right clicks on the GUIHAR object and chooses the “Blank” menu item. The GUIHAR object creates a CommandStatus and then calls the CHART2HAR served object which the GUIHAR wraps. If successful, the server will push events indicating the changed status.

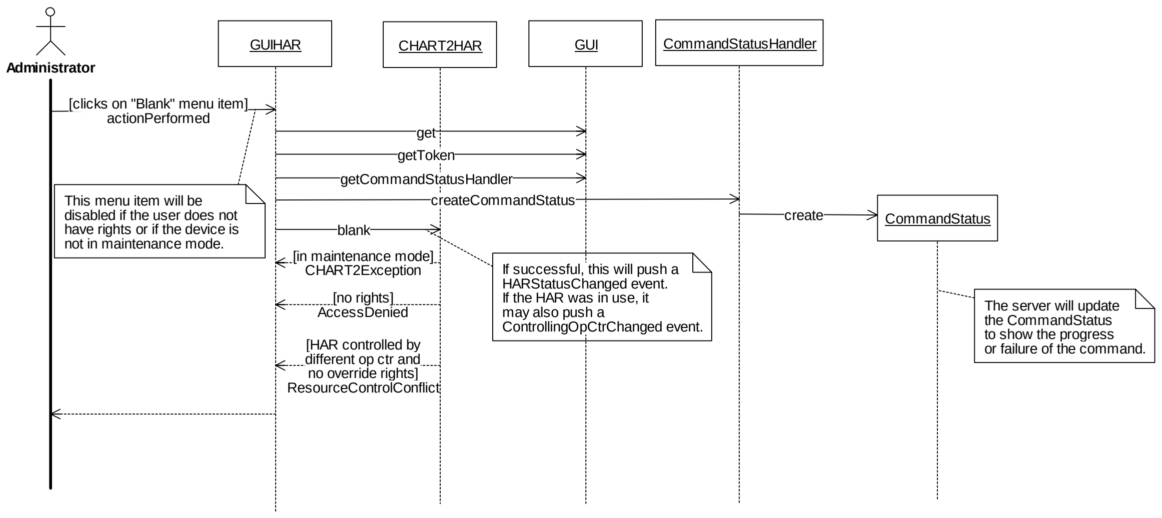
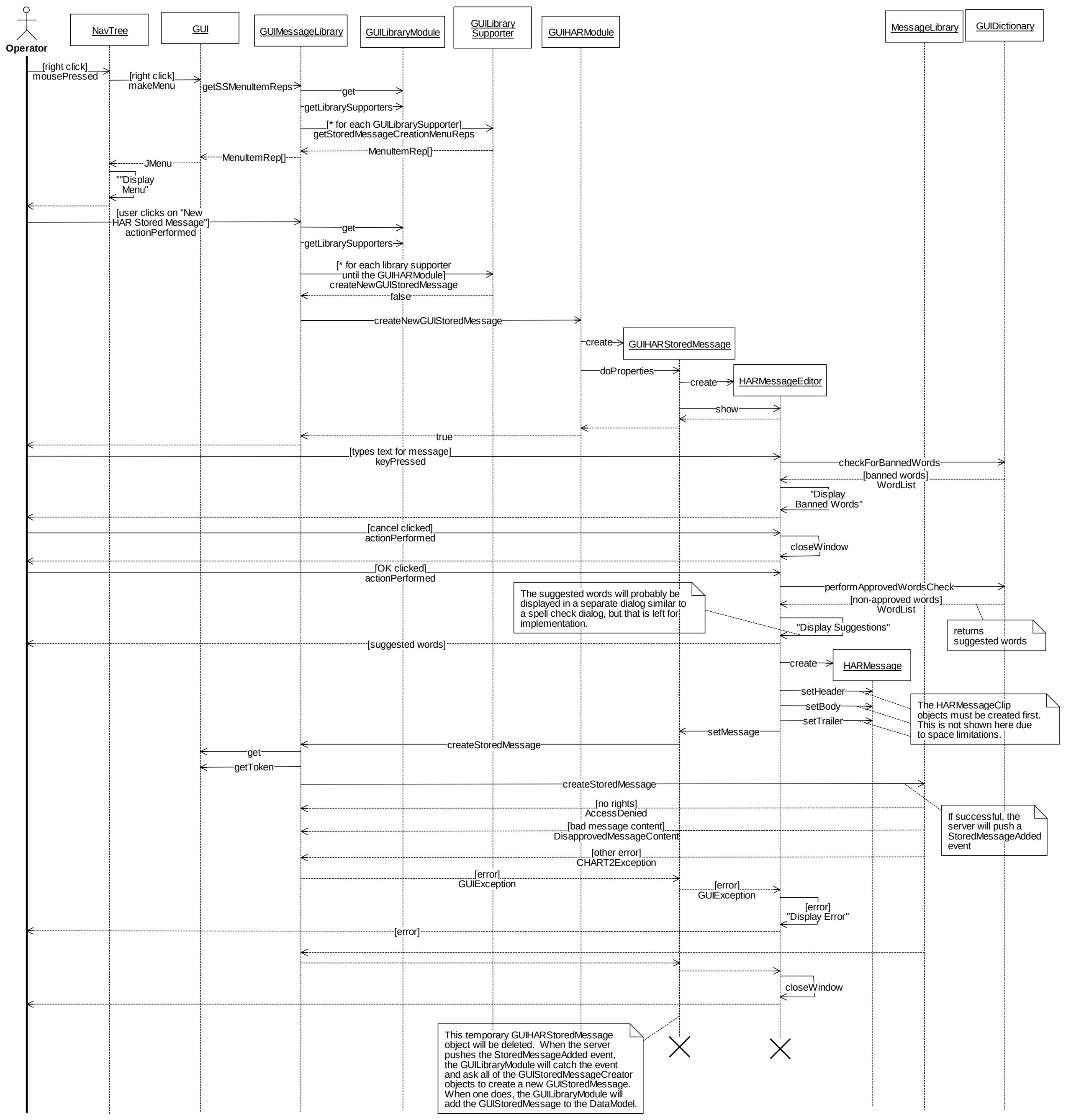

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Blank" menu item]
actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is not in maintenance mode.
blank
[in maintenance mode]
CHART2Exception
If successful, this will push a HARStatusChanged event. If the HAR was in use, it may also push a ControllingOpCtrChanged event.
[no rights]
AccessDenied
The server will update the CommandStatus to show the progress or failure of the command.
[HAR controlled by different op ctr and no override rights]
ResourceControlConflict

Figure 69. GUIHARModule:BlankHARInMaintenanceMode (Sequence Diagram)

3.8.2.7 GUIHARModule:CreateHARStoredMessage (Sequence Diagram)

This diagram shows how a HAR stored message is created. First, the user right clicks on the GUIMessageLibrary object, which calls the GUILibraryModule to get the installed message creators. Each message creator returns menu items for message types that it can create. When the user clicks on the appropriate message type, the GUIMessageLibrary object is called again, and this time it asks each message creator to create the correct type of message based on the menu item. The GUIHARModule creates a temporary GUIHARStoredMessage object to edit, and calls doProperties to show the HARMessageEditor dialog. As the user types, any banned words will be shown to the user. When the user clicks OK, the non-approved words will be displayed to the user. Once the results of the approved words check are accepted by the user, the message editor will create a new HARMessage object and will call setMessage on the GUIHARStoredMessage wrapper object. Since the wrapper does not contain a served StoredMessage object, it calls the message library to create one. If successful, the server will create a new StoredMessage object and will push an event to update all of the GUIs.



The suggested words will probably be displayed in a separate dialog similar to a spell check dialog, but that is left for implementation.

returns suggested words

The HARMessageClip objects must be created first. This is not shown here due to space limitations.

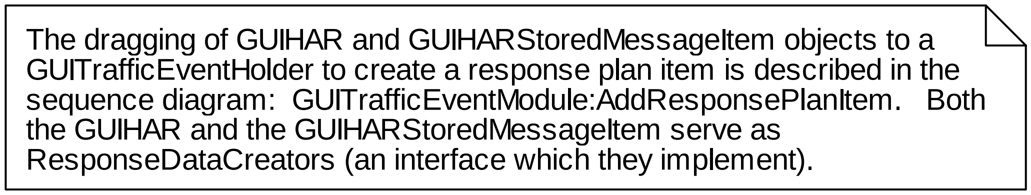
If successful, the server will push a StoredMessageAdded event

This temporary GUIHARStoredMessage object will be deleted. When the server pushes the StoredMessageAdded event, the GUILibraryModule will catch the event and ask all of the GUIStoredMessageCreator objects to create a new GUIStoredMessage. When one does, the GUILibraryModule will add the GUIStoredMessage to the DataModel.

Figure 70. GUIHARModule:CreateHARStoredMessage (Sequence Diagram)

3.8.2.8 GUIHARModule:CreateResponsePlanItem (Sequence Diagram)

This diagram shows how a HAR response plan item is added to the system. The user drags a GUIHAR or a GUIPlanItem object over the GUITrafficEventHolder (the object representing the traffic event in the GUI) and drops it. Since the GUIHAR and GUIHARMsgItem objects both implement the ResponseDataCreator interface, the GUITrafficEventModule uses either of these to create a HARResponsePlanItemData, which it then uses to create a ResponsePlanItem. See the sequence diagram: GUITrafficEventModule:AddResponsePlanItem for details.



The dragging of GUIHAR and GUIHARStoredMessageItem objects to a GUITrafficEventHolder to create a response plan item is described in the sequence diagram: GUITrafficEventModule:AddResponsePlanItem. Both the GUIHAR and the GUIHARStoredMessageItem serve as ResponseDataCreators (an interface which they implement).

Figure 171. GUIHARModule:CreateResponsePlanItem (Sequence Diagram)

3.8.2.9 GUIHARModule:DeleteHARMessageFromController (Sequence Diagram)

This diagram shows how a message is deleted from a HAR controller's slot. The administrator does this from the HAR Properties Dialog, and clicks on the "Delete" button when viewing the slot contents. The dialog calls the GUIHAR wrapper object to delete the message, which in turn calls the CHART2HAR object that it wraps (after creating a CommandStatus object).

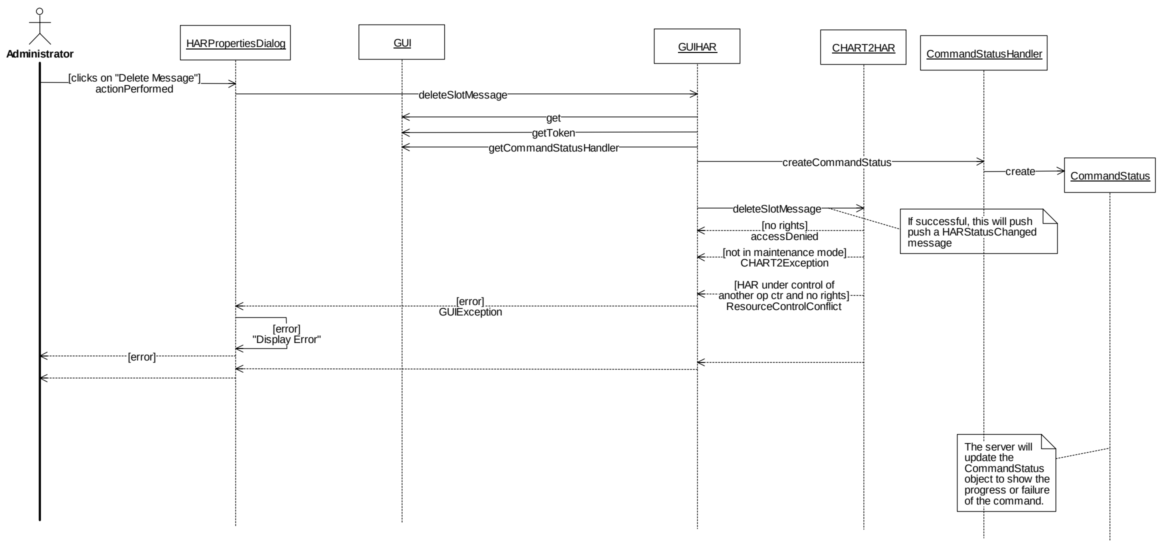
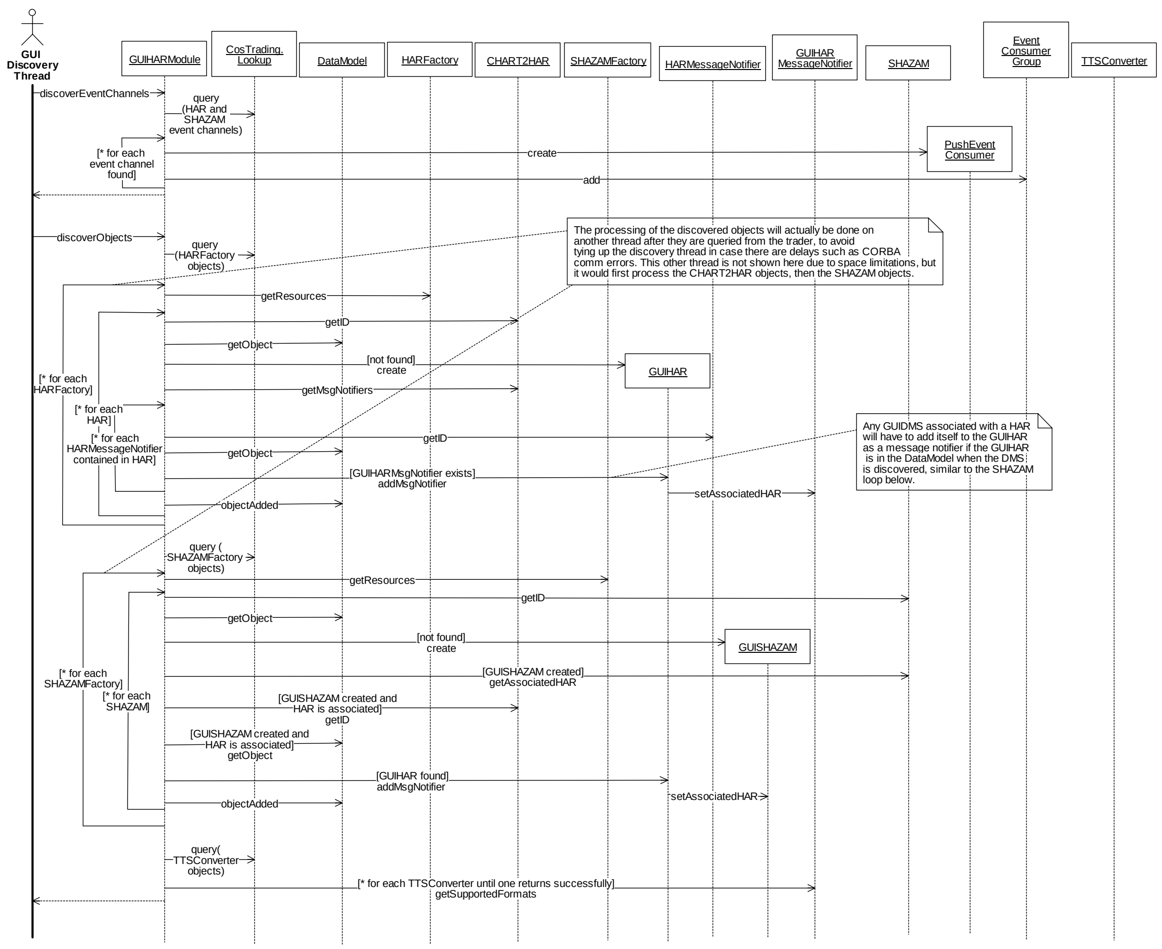


Figure 72. GUIHARModule:DeleteHARMessageFromController (Sequence Diagram)

3.8.2.10 GUIHARModule:Discovery (Sequence Diagram)

This diagram shows the event channel and object discovery, which is done after startup and periodically thereafter. In event channel discovery, the module queries the event channels from the trading service and creates a PushConsumer to receive the CORBA events, then adds each to the EventConsumerGroup for maintenance of the event channel. In object discovery, the HAR module looks for any HARFactory objects in the trader, and asks for all of the HARs served by each factory. A GUIHAR wrapper object is created and added to the DataModel, and the GUIHARMessageNotifiers are associated to the HAR (if the message notifiers are already in the DataModel). The HAR module then gets the SHAZAMFactory objects from the trader and gets all of the SHAZAM objects served by each factory. Then it associates the GUISHAZAM objects with the GUIHAR objects (if appropriate and the GUIHAR object is in the DataModel). The HAR module also queries the TTSConverter objects from the trader, to call when text-to-speech conversion is required.


GUI Discovery Thread
GUIHARModule
CosTrading. Lookup
DataModel
HARFactory
CHART2HAR
SHAZAMFactory
HARMessageNotifier
GUIHAR MessageNotifier
SHAZAM
Event Consumer Group
TTSConverter
discoverEventChannels
query (HAR and SHAZAM event channels)
[* for each event channel found]
create
PushEvent Consumer
add
discoverObjects
query (HARFactory objects)
getResources
getID
getObject
[not found] create
GUIHAR
getMsgNotifiers
[* for each HARFactory]
[* for each HAR]
[* for each HARMessageNotifier contained in HAR]
getID
getObject
[GUIHARMsgNotifier exists] addMsgNotifier
setAssociatedHAR
objectAdded
query (SHAZAMFactory objects)
getResources
getID
getObject
[not found] create
GUISHAZAM
[GUISHAZAM created] getAssociatedHAR
[* for each SHAZAMFactory]
[* for each SHAZAM]
[GUISHAZAM created and HAR is associated] getID
[GUISHAZAM created and HAR is associated] getObject
[GUIHAR found] addMsgNotifier
setAssociatedHAR
objectAdded
query(TTSConverter objects)
[* for each TTSConverter until one returns successfully] getSupportedFormats

The processing of the discovered objects will actually be done on another thread after they are queried from the trader, to avoid tying up the discovery thread in case there are delays such as CORBA comm errors. This other thread is not shown here due to space limitations, but it would first process the CHART2HAR objects, then the SHAZAM objects.

Any GUIDMS associated with a HAR will have to add itself to the GUIHAR as a message notifier if the GUIHAR is in the DataModel when the DMS is discovered, similar to the SHAZAM loop below.

Figure 73. Discovery:Basic (Sequence Diagram)

3.8.2.11 GUIHARModule:ListenToAudioClip (Sequence Diagram)

This diagram shows how an audio clip is played from the GUI. The user clicks on the play button in the message editor or HAR properties dialog, and the dialog creates an AudioPushConsumerImpl, activates the object via the CORBA POA, and calls the HARMessageAudioClip to stream the message. The server will then call the AudioPushConsumerImpl to report the format of the audio and to stream the chunks of data. The AudioPushConsumerImpl will call the dialog, as it implements the AudioPushListener interface. The dialog will open, write to, and close the SourceDataLine that represents the audio output.

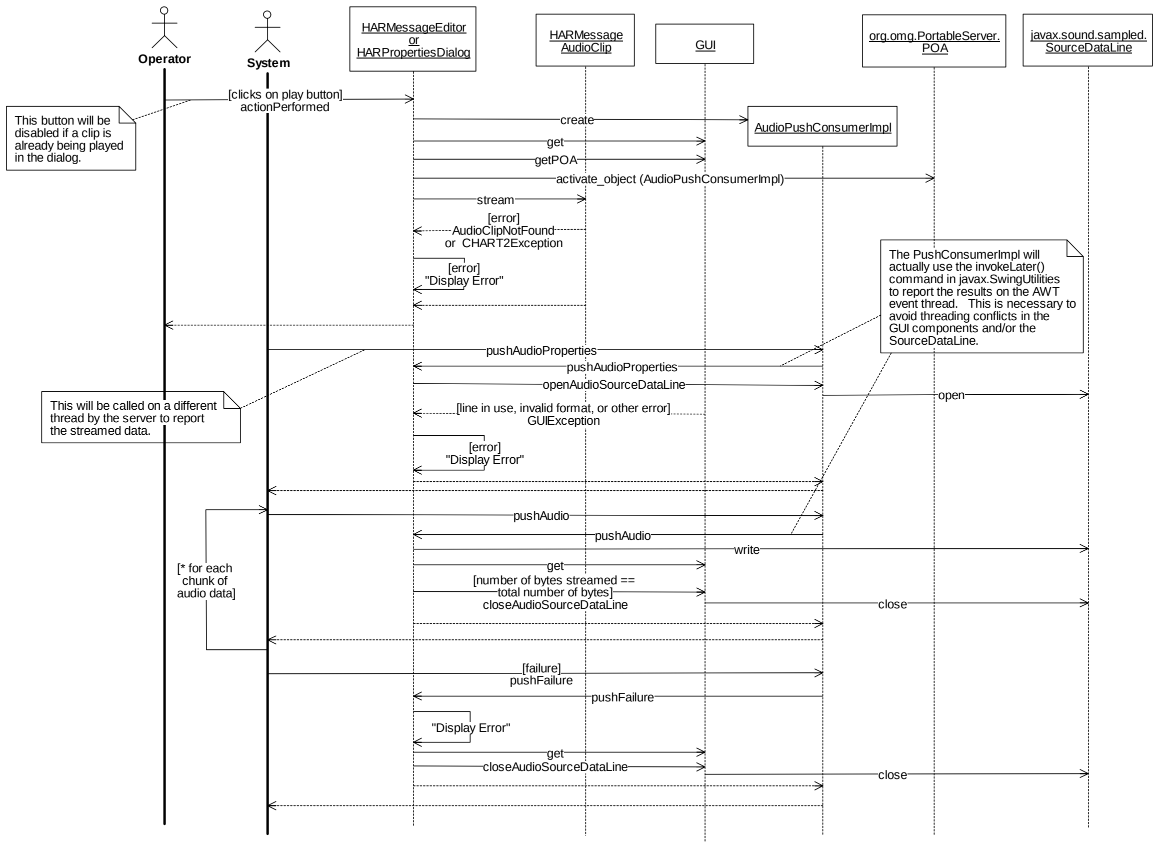
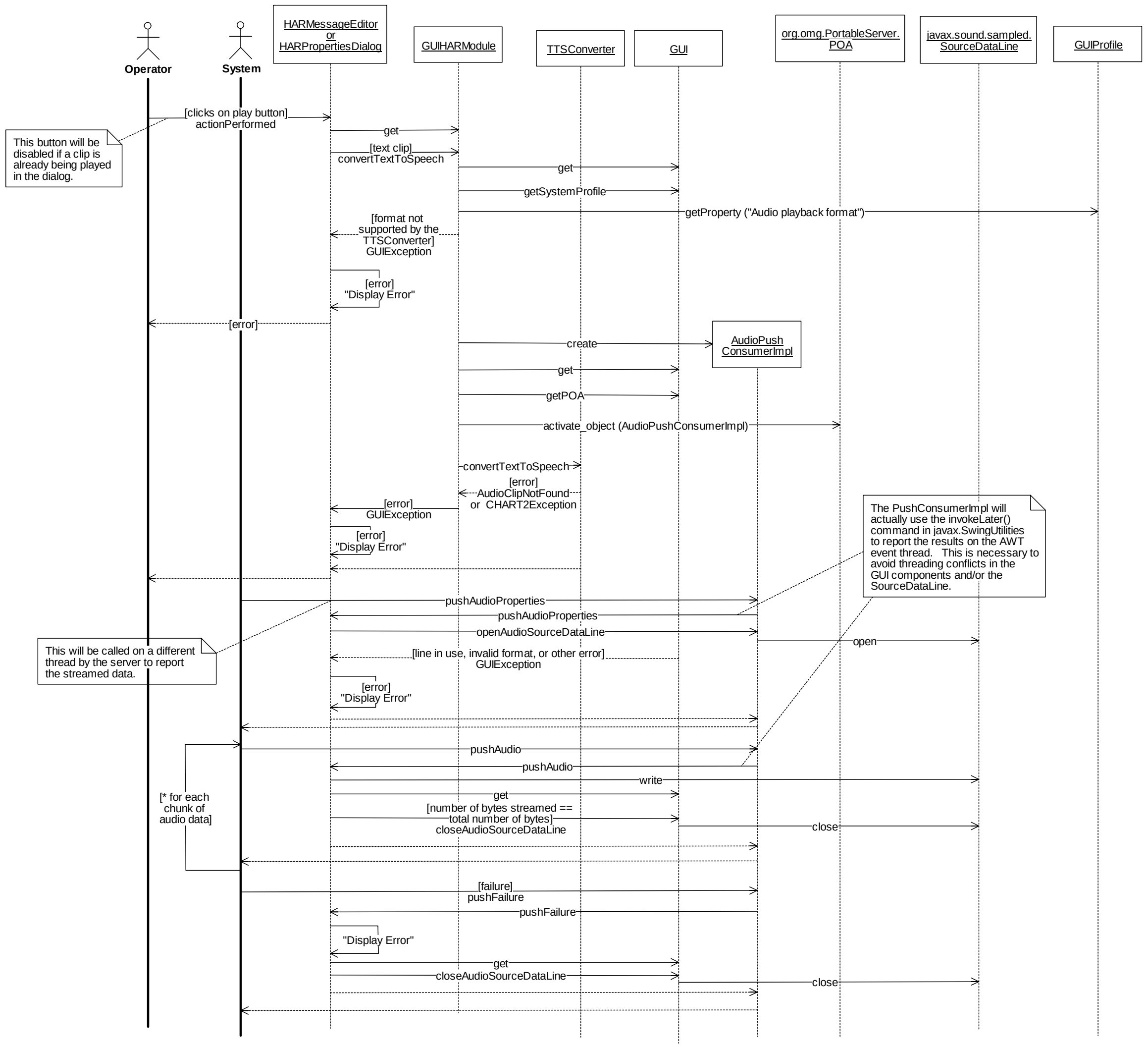

Operator
System
HARMessageEditor or HARPropertiesDialog
HARMessage AudioClip
GUI
org.omg.PortableServer. POA
javax.sound.sampled. SourceDataLine
AudioPushConsumerImpl
This button will be disabled if a clip is already being played in the dialog.
[clicks on play button] actionPerformed
create
get
getPOA
activate_object (AudioPushConsumerImpl)
stream
[error] AudioClipNotFound or CHART2Exception
[error] "Display Error"
The PushConsumerImpl will actually use the invokeLater() command in javax.SwingUtilities to report the results on the AWT event thread. This is necessary to avoid threading conflicts in the GUI components and/or the SourceDataLine.
pushAudioProperties
pushAudioProperties
openAudioSourceDataLine
open
This will be called on a different thread by the server to report the streamed data.
[line in use, invalid format, or other error] GUIException
[error] "Display Error"
pushAudio
pushAudio
write
[* for each chunk of audio data]
get
[number of bytes streamed == total number of bytes] closeAudioSourceDataLine
close
[failure] pushFailure
pushFailure
"Display Error"
get
closeAudioSourceDataLine
close

Figure 74. GUIHARModule:ListenToAudioClip (Sequence Diagram)

3.8.2.12 GUIHARModule:ListenToTextClip (Sequence Diagram)

This diagram shows how text is played as audio for the user for review. From the HAR Message Editor or HAR Properties Dialog, the user would click on the "Play" button. If a text clip is being played, the dialog would get the text from the clip. Then it would create an AudioPushConsumerImpl to listen for the results, then call the GUIHARModule to convert to text. The module would get the playback format stored in the system profile (or use the default format if no playback format property is found), and call the TTSConverter to convert the text to speech. The system would then call back to the AudioPushConsumerImpl to stream the data. The AudioPushConsumerImpl would then cause the AudioPushListener (i.e., the dialog) to be called on the main thread to report the results.



Operator
System
HARMessageEditor or HARPropertiesDialog
GUIHARModule
TTSConverter
GUI
org.omg.PortableServer.POA
javax.sound.sampled.SourceDataLine
GUIProfile
[clicks on play button] actionPerformed
This button will be disabled if a clip is already being played in the dialog.
get
[text clip] convertTextToSpeech
get
getSystemProfile
getProperty ("Audio playback format")
[format not supported by the TTSConverter] GUIException
[error] "Display Error"
[error]
create
AudioPush ConsumerImpl
get
getPOA
activate_object (AudioPushConsumerImpl)
convertTextToSpeech
[error] AudioClipNotFound or CHART2Exception
[error] GUIException
[error] "Display Error"
The PushConsumerImpl will actually use the invokeLater() command in javax.SwingUtilities to report the results on the AWT event thread.   This is necessary to avoid threading conflicts in the GUI components and/or the SourceDataLine.
pushAudioProperties
pushAudioProperties
openAudioSourceDataLine
open
This will be called on a different thread by the server to report the streamed data.
[line in use, invalid format, or other error] GUIException
[error] "Display Error"
pushAudio
pushAudio
write
[* for each chunk of audio data]
get
[number of bytes streamed == total number of bytes] closeAudioSourceDataLine
close
[failure] pushFailure
pushFailure
"Display Error"
get
closeAudioSourceDataLine
close

Figure 75. GUIHARModule:ListenToTextClip (Sequence Diagram)

3.8.2.13 Login:Basic (Sequence Diagram)

This diagram shows what happens during login. The GUI calls each InstallableModule's loggedIn() method, but the GUIHARModule does not do any work at login.

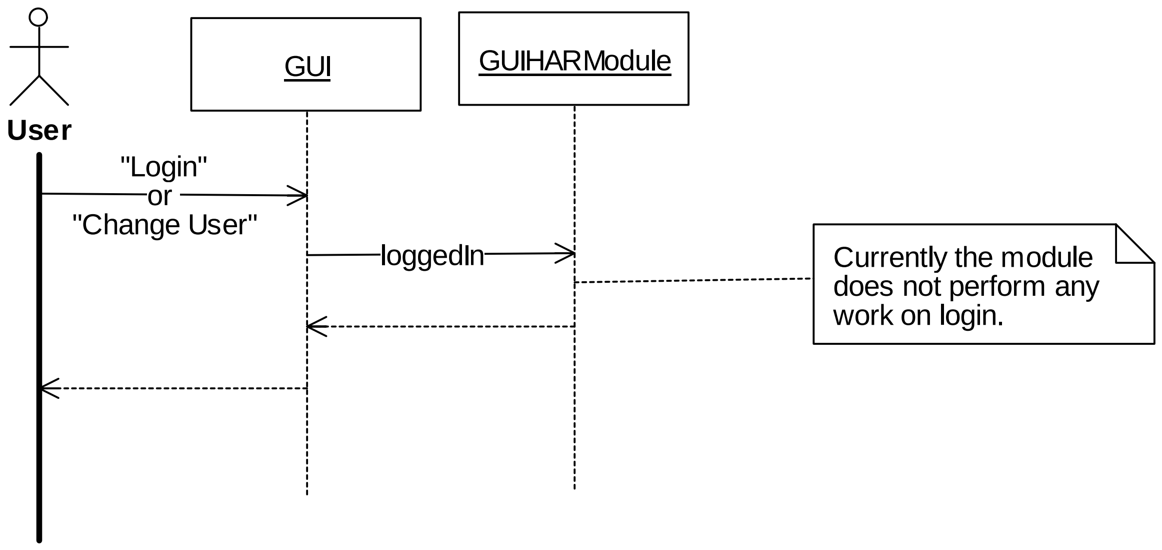


Figure 76. Login:Basic (Sequence Diagram)

3.8.2.14 GUIHARModule:Logout (Sequence Diagram)

This diagram shows what happens when the user logs out. The GUI calls all of the InstallableModule objects' loggedOut() methods, but the GUIHARModule currently does nothing during logout.

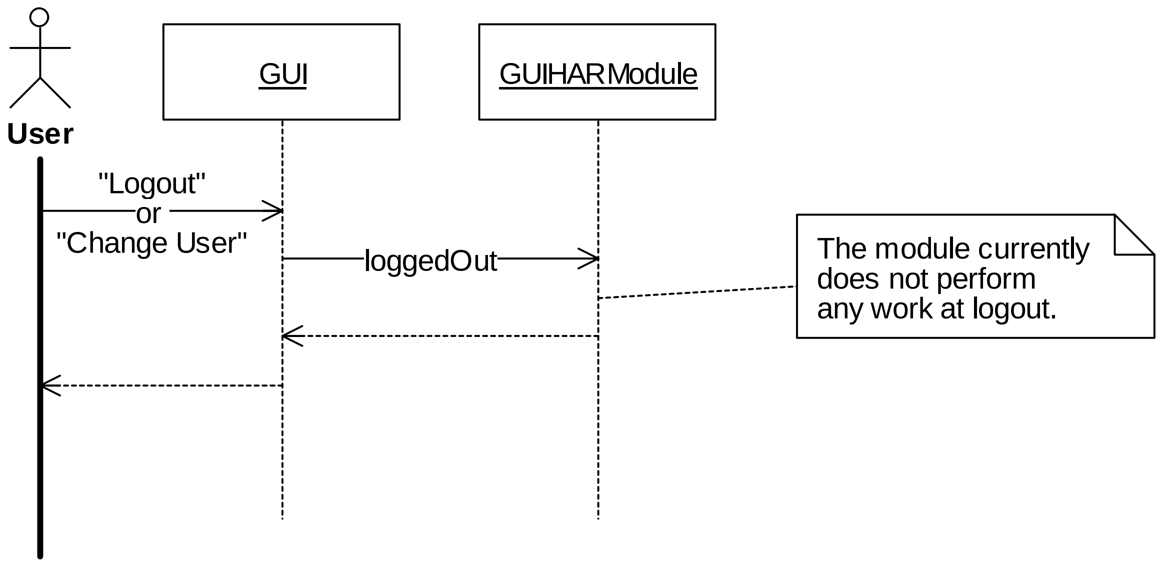

User
GUI
GUIHARModule
"Logout" or "Change User"
loggedOut
The module currently does not perform any work at logout.

Figure 77. GUIHARModule:Logout (Sequence Diagram)

3.8.2.15 GUIHARModule:ModifyHARSettings (Sequence Diagram)

This diagram shows how the HAR settings are modified. The user right clicks on the GUIHAR object in the Navigator and clicks on the “Properties” menu item. The GUIHAR object then creates a HARPropertiesDialog, which calls back to the GUIHAR to get the configuration and slot usage to initialize itself with. The GUIHAR wrapper object then calls the CHART2HAR object in the server to get this information. After the administrator is done editing the configuration, clicking on the “OK” button will cause the dialog to call the GUIHAR object’s setConfiguration() method. The GUIHAR will create a CommandStatus object and will call the CHART2HAR that it wraps to set the configuration. If successful, the server will push a CORBA event indicating that the configuration has changed.

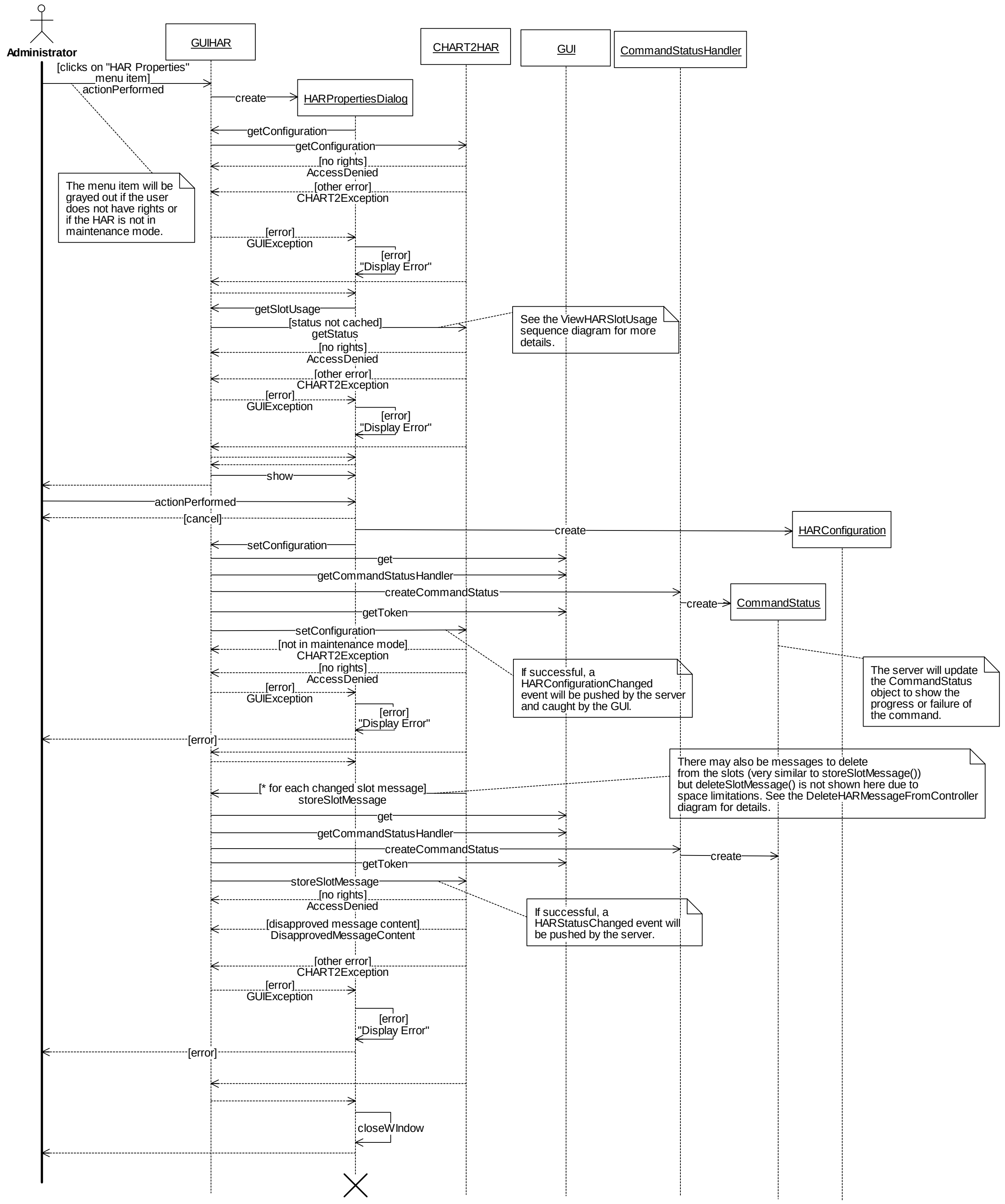

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "HAR Properties" menu item]
actionPerformed
create
HARPropertiesDialog
getConfiguration
getConfiguration
[no rights]
AccessDenied
[other error]
CHART2Exception
The menu item will be grayed out if the user does not have rights or if the HAR is not in maintenance mode.
[error]
GUIException
[error]
"Display Error"
getSlotUsage
[status not cached]
getStatus
See the ViewHARSlotUsage sequence diagram for more details.
[no rights]
AccessDenied
[other error]
CHART2Exception
[error]
GUIException
[error]
"Display Error"
show
actionPerformed
[cancel]
create
HARConfiguration
setConfiguration
get
getCommandStatusHandler
createCommandStatus
create
CommandStatus
getToken
setConfiguration
[not in maintenance mode]
CHART2Exception
[no rights]
AccessDenied
[error]
GUIException
[error]
"Display Error"
If successful, a HARConfigurationChanged event will be pushed by the server and caught by the GUI.
The server will update the CommandStatus object to show the progress or failure of the command.
[error]
There may also be messages to delete from the slots (very similar to storeSlotMessage()) but deleteSlotMessage() is not shown here due to space limitations. See the DeleteHARMessageFromController diagram for details.
[* for each changed slot message]
storeSlotMessage
get
getCommandStatusHandler
createCommandStatus
create
getToken
storeSlotMessage
[no rights]
AccessDenied
If successful, a HARStatusChanged event will be pushed by the server.
[disapproved message content]
DisapprovedMessageContent
[other error]
CHART2Exception
[error]
GUIException
[error]
"Display Error"
[error]
closeWIndow

Figure 78. GUIHARModule:ModifyHARSettings (Sequence Diagram)

3.8.2.16 GUIHARModule:ModifyHARStoredMessage (Sequence Diagram)

This diagram shows how the contents of a stored message are modified. The user clicks on an existing GUIHARStoredMessage object in the Navigator, and clicks on the “Properties” menu item. The GUIHARStoredMessage then invokes the HARMessageEditor dialog. On initialization, the dialog calls back to the GUIHARStoredMessage wrapper object to get the message content, which calls back to the StoredMessage object in the server if the message is not already cached in the wrapper object. When the HARMessage is returned, the dialog can be initialized from the existing message contents. As the user types in text for the message, the banned words will be displayed. When the user clicks “OK”, the dialog first checks the non-approved words and displays them, or if all words are approved, it calls the GUIHARModule to create the audio or text clips for the header, body, and trailer of the message. These clips are then set into the HARMessage, and the dialog calls the GUIHARStoredMessage to set the message, which in turn calls the StoredMessage object in the server. If successful, the server will push a CORBA event to update the GUIs.

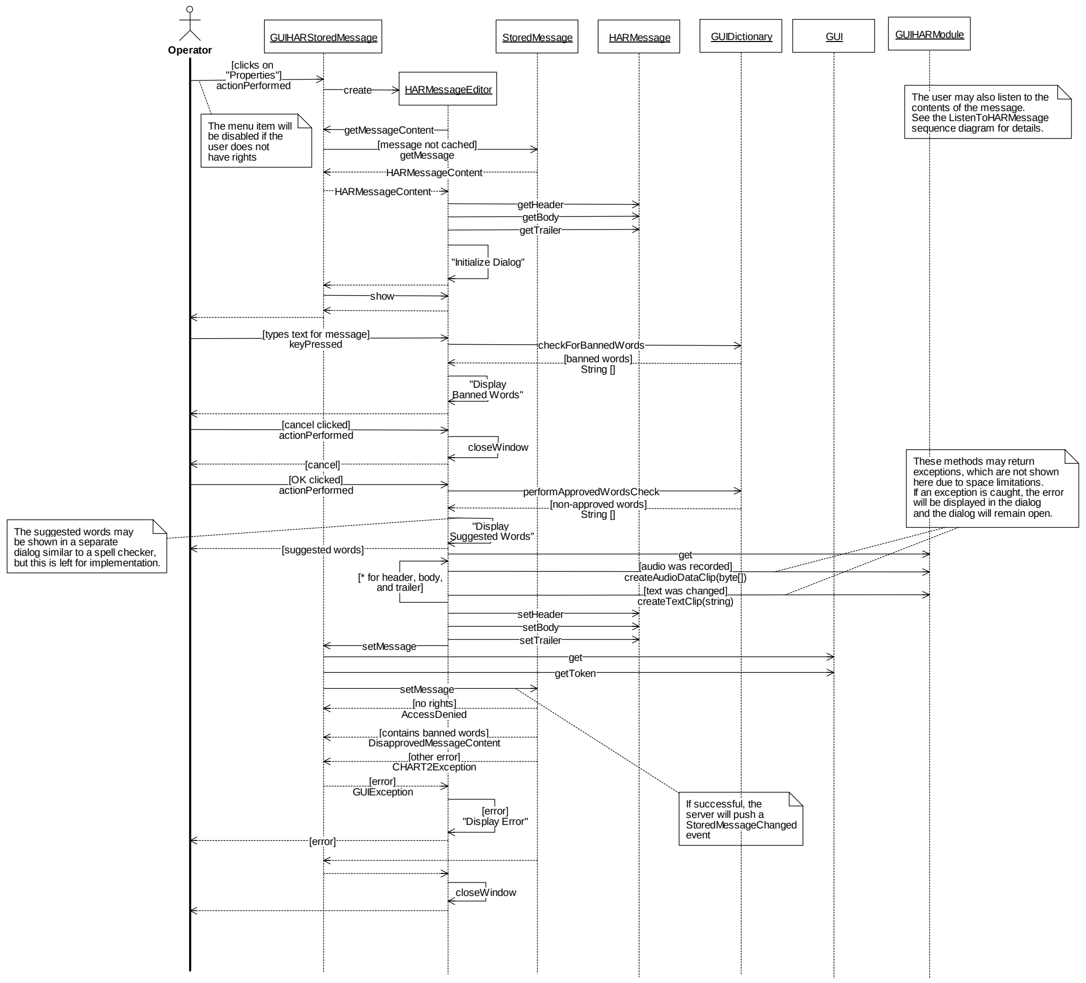

Operator
GUIHARStoredMessage
StoredMessage
HARMessage
GUIDictionary
GUI
GUIHARModule
[clicks on "Properties"] actionPerformed
create
HARMessageEditor
The user may also listen to the contents of the message. See the ListenToHARMessage sequence diagram for details.
The menu item will be disabled if the user does not have rights
getMessageContent
[message not cached] getMessage
HARMessageContent
HARMessageContent
getHeader
getBody
getTrailer
"Initialize Dialog"
show
[types text for message] keyPressed
checkForBannedWords
[banned words] String []
"Display Banned Words"
[cancel clicked] actionPerformed
closeWindow
[cancel]
These methods may return exceptions, which are not shown here due to space limitations. If an exception is caught, the error will be displayed in the dialog and the dialog will remain open.
[OK clicked] actionPerformed
performApprovedWordsCheck
[non-approved words] String []
The suggested words may be shown in a separate dialog similar to a spell checker, but this is left for implementation.
"Display Suggested Words"
[suggested words]
get
[* for header, body, and trailer]
[audio was recorded] createAudioDataClip(byte[])
[text was changed] createTextClip(string)
setHeader
setBody
setTrailer
setMessage
get
getToken
setMessage
[no rights] AccessDenied
[contains banned words] DisapprovedMessageContent
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
If successful, the server will push a StoredMessageChanged event
[error]
closeWindow

Figure 79. GUIHARModule:ModifyHARStoredMessage (Sequence Diagram)

3.8.2.17 GUIHARModule:ModifySHAZAMSettings (Sequence Diagram)

This diagram shows how the SHAZAM settings are modified. The user right clicks on the GUISHAZAM object in the Navigator and clicks on the “Properties” menu item. The GUISHAZAM object then creates a SHAZAMPropertiesDialog, which calls back to the GUISHAZAM to get the configuration and slot usage to initialize itself with. The GUISHAZAM wrapper object then calls the SHAZAM object in the server to get this information if it is not already cached. After the administrator is done editing the configuration, clicking on the “OK” button will cause the dialog to call the GUISHAZAM object’s setConfiguration() method. The GUISHAZAM will create a CommandStatus object and will call the SHAZAM that it wraps to set the configuration. If successful, the server will push a CORBA event indicating that the configuration has changed.

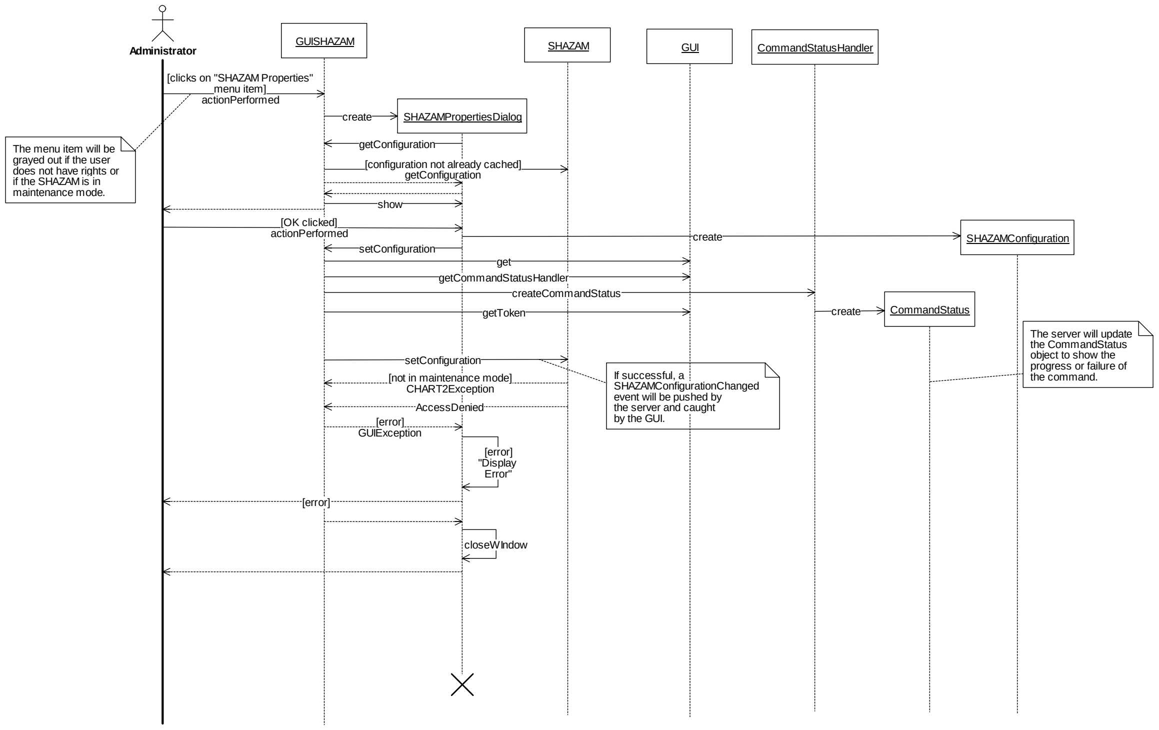

Administrator
GUISHAZAM
SHAZAM
GUI
CommandStatusHandler
[clicks on "SHAZAM Properties" menu item] actionPerformed
create
SHAZAMPropertiesDialog
The menu item will be grayed out if the user does not have rights or if the SHAZAM is in maintenance mode.
getConfiguration
[configuration not already cached] getConfiguration
show
[OK clicked] actionPerformed
create
SHAZAMConfiguration
setConfiguration
get
getCommandStatusHandler
createCommandStatus
create
CommandStatus
getToken
The server will update the CommandStatus object to show the progress or failure of the command.
setConfiguration
If successful, a SHAZAMConfigurationChanged event will be pushed by the server and caught by the GUI.
[not in maintenance mode] CHART2Exception
AccessDenied
[error] GUIException
[error] "Display Error"
[error]
closeWindow

Figure 80. GUIHARModule:ModifySHAZAMSettings (Sequence Diagram)

3.8.2.18 GUIHARModule:PutHARInMaintenanceMode (Sequence Diagram)

This diagram shows how a HAR is put into maintenance mode. The Administrator right clicks on a GUIHAR in the Navigator and clicks on the “Put In Maintenance Mode” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the CHART2HAR object (which it wraps) to put it in maintenance mode. If successful, the server will push a CORBA event indicating that the comm mode has been changed.

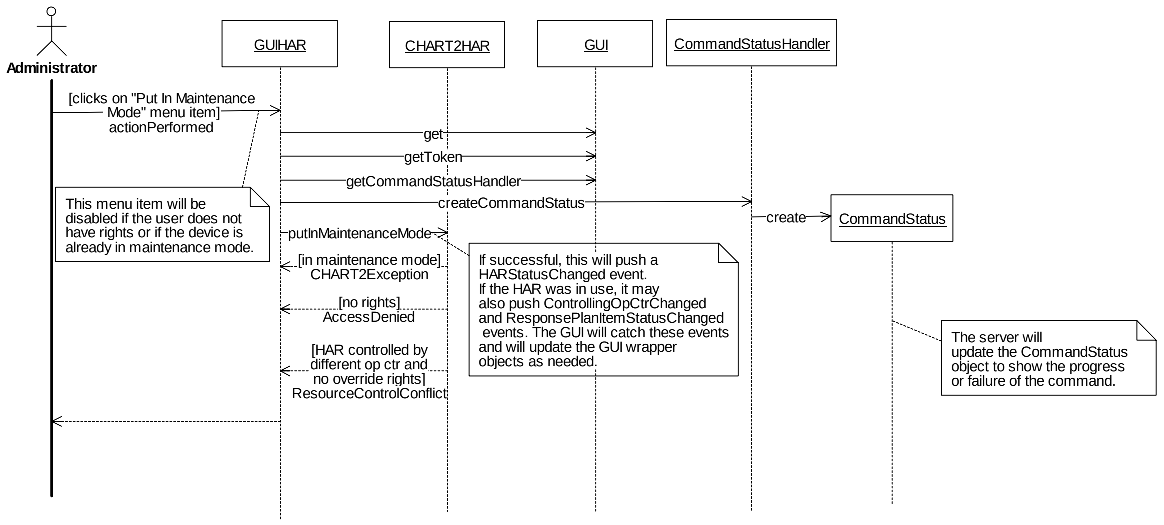

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Put In Maintenance Mode" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is already in maintenance mode.
putInMaintenanceMode
[in maintenance mode] CHART2Exception
[no rights] AccessDenied
[HAR controlled by different op ctr and no override rights] ResourceControlConflict
If successful, this will push a HARStatusChanged event. If the HAR was in use, it may also push ControllingOpCtrChanged and ResponsePlanItemStatusChanged events. The GUI will catch these events and will update the GUI wrapper objects as needed.
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 81. GUIHARModule:PutHARInMaintenanceMode (Sequence Diagram)

3.8.2.19 GUIHARModule:PutHAROnline (Sequence Diagram)

This diagram shows how a HAR is put online. The Administrator right clicks on a GUIHAR in the Navigator and clicks on the “Put Online” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the CHART2HAR object (which it wraps) to put it online. If successful, the server will push a CORBA event indicating that the comm mode has been changed.

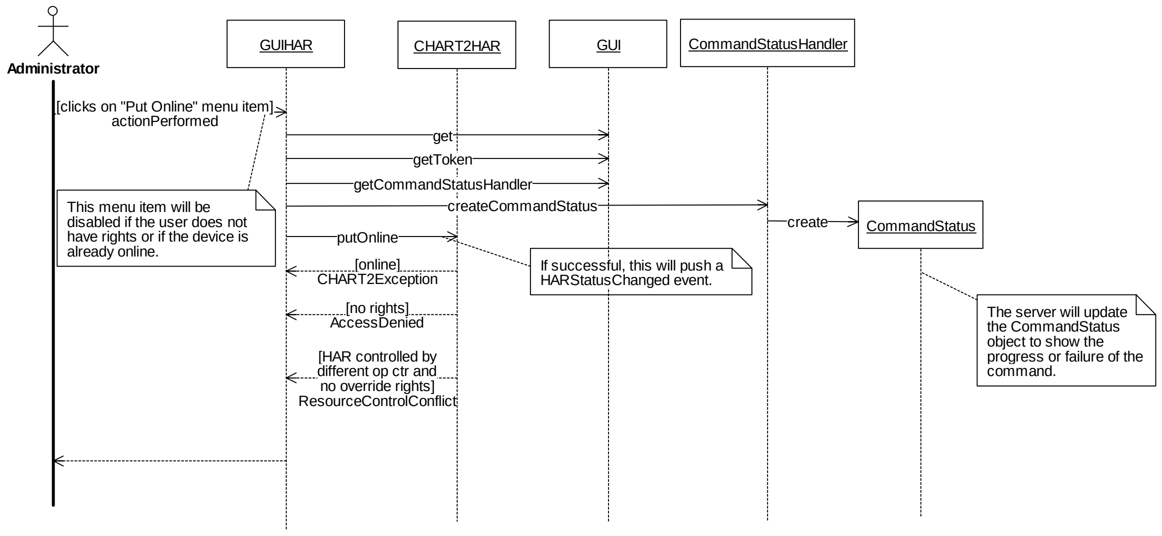

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Put Online" menu item]
actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is already online.
putOnline
[online]
CHART2Exception
If successful, this will push a HARStatusChanged event.
[no rights]
AccessDenied
The server will update the CommandStatus object to show the progress or failure of the command.
[HAR controlled by different op ctr and no override rights]
ResourceControlConflict

Figure 82. GUIHARModule:PutHAROnline (Sequence Diagram)

3.8.2.20 GUIHARModule:PutSHAZAMInMaintenanceMode (Sequence Diagram)

This diagram shows how a SHAZAM is put into maintenance mode. The Administrator right clicks on a GUISHAZAM object in the Navigator and clicks on the “Put In Maintenance Mode” menu item. The GUISHAZAM creates a CommandStatus object to monitor the progress of the command and calls the SHAZAM object (which it wraps) to put it in maintenance mode. If successful, the server will push a CORBA event indicating that the comm mode has been changed.

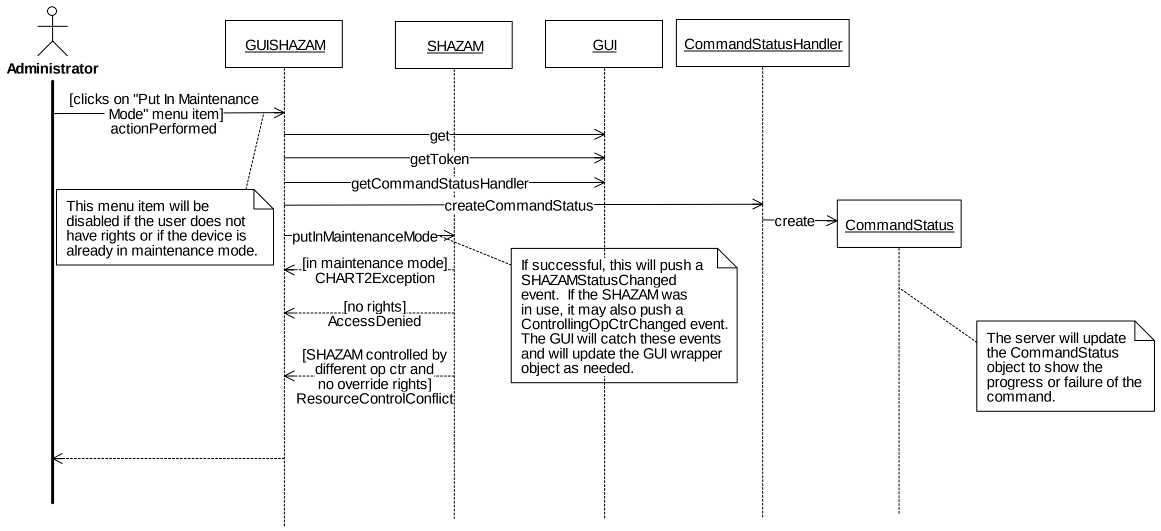
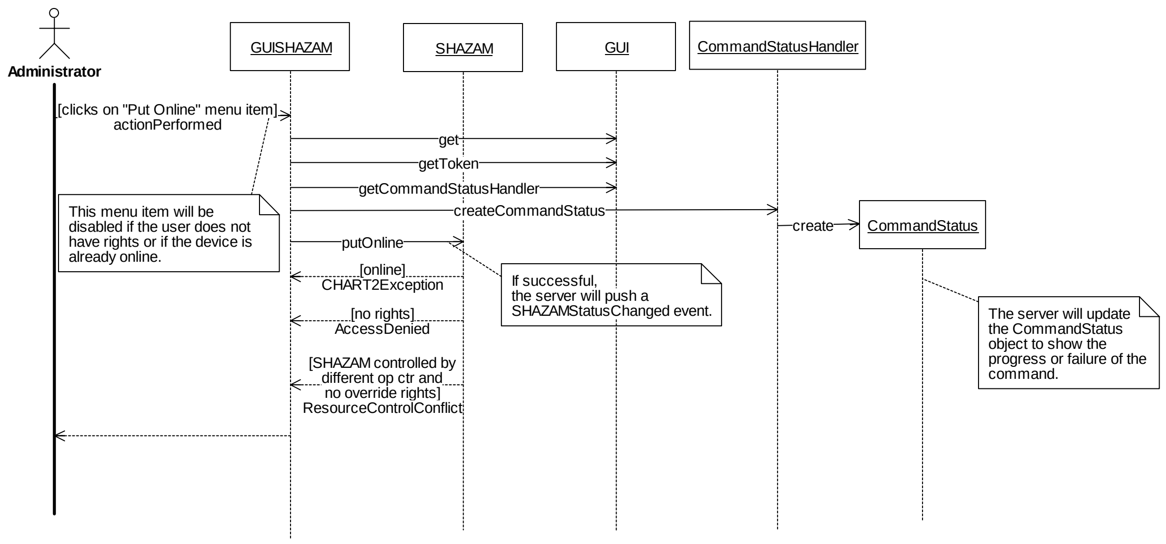

Administrator
GUISHAZAM
SHAZAM
GUI
CommandStatusHandler
[clicks on "Put In Maintenance Mode" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is already in maintenance mode.
putInMaintenanceMode
[in maintenance mode] CHART2Exception
[no rights] AccessDenied
[SHAZAM controlled by different op ctr and no override rights] ResourceControlConflict
If successful, this will push a SHAZAMStatusChanged event. If the SHAZAM was in use, it may also push a ControllingOpCtrChanged event. The GUI will catch these events and will update the GUI wrapper object as needed.
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 83. GUIHARModule:PutSHAZAMInMaintenanceMode (Sequence Diagram)

3.8.2.21 GUIHARModule:PutSHAZAMOnline (Sequence Diagram)

This diagram shows how a SHAZAM is put online. The Administrator right clicks on a GUISHAZAM object in the Navigator and clicks on the “Put Online” menu item. The GUISHAZAM creates a CommandStatus object to monitor the progress of the command and calls the SHAZAM object (which it wraps) to put it online. If successful, the server will push a CORBA event indicating that the comm mode has been changed.



Administrator
GUISHAZAM
SHAZAM
GUI
CommandStatusHandler
[clicks on "Put Online" menu item]
actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is already online.
putOnline
[online]
CHART2Exception
If successful, the server will push a SHAZAMStatusChanged event.
[no rights]
AccessDenied
[SHAZAM controlled by different op ctr and no override rights]
ResourceControlConflict
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 84. GUIHARModule:PutSHAZAMOnline (Sequence Diagram)

3.8.2.22 GUIHARModule:RemoveHAR (Sequence Diagram)

This diagram shows how a HAR is removed from the system. The Administrator right clicks on a GUIHAR object in the Navigator and clicks on the “Remove HAR” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the remove() method of the CHART2HAR object (which it wraps). If successful, the server will push a CORBA event indicating that the HAR was removed.

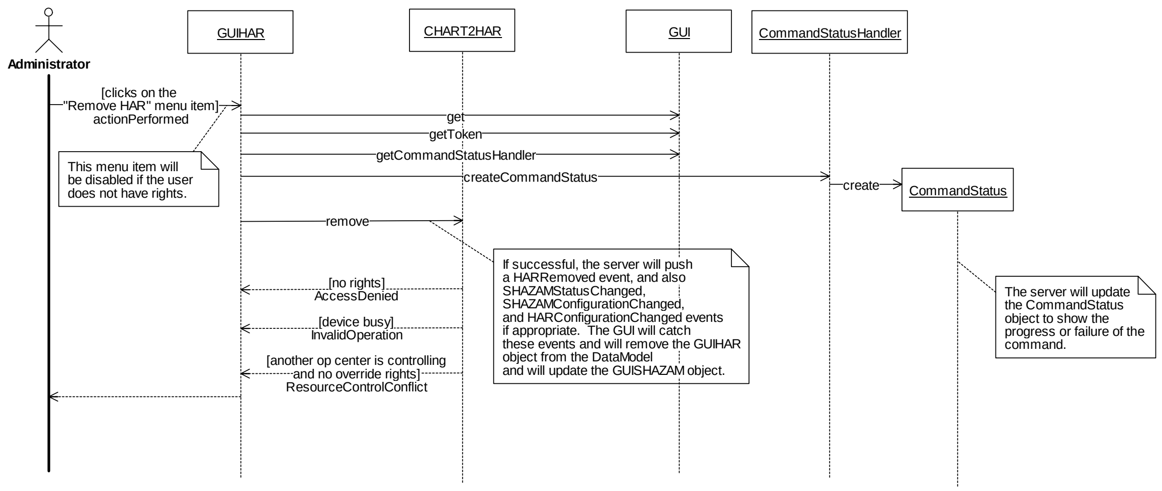
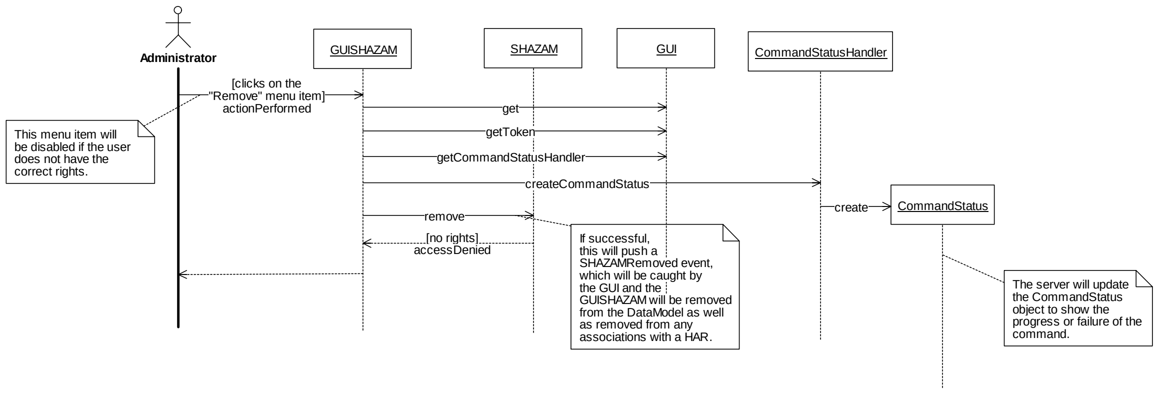

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on the "Remove HAR" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights.
remove
If successful, the server will push a HARRemoved event, and also SHAZAMStatusChanged, SHAZAMConfigurationChanged, and HARConfigurationChanged events if appropriate. The GUI will catch these events and will remove the GUIHAR object from the DataModel and will update the GUISHAZAM object.
[no rights] AccessDenied
[device busy] InvalidOperation
[another op center is controlling and no override rights] ResourceControlConflict
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 85. GUIHARModule:RemoveHAR (Sequence Diagram)

3.8.2.23 GUIHARModule:RemoveSHAZAM (Sequence Diagram)

This diagram shows how a SHAZAM is removed from the system. The Administrator right clicks on a GUISHAZAM object in the Navigator and clicks on the “Remove SHAZAM” menu item. The GUISHAZAM creates a CommandStatus object to monitor the progress of the command and calls the remove() method of the SHAZAM object (which it wraps). If successful, the server will push a CORBA event indicating that the SHAZAM was removed.

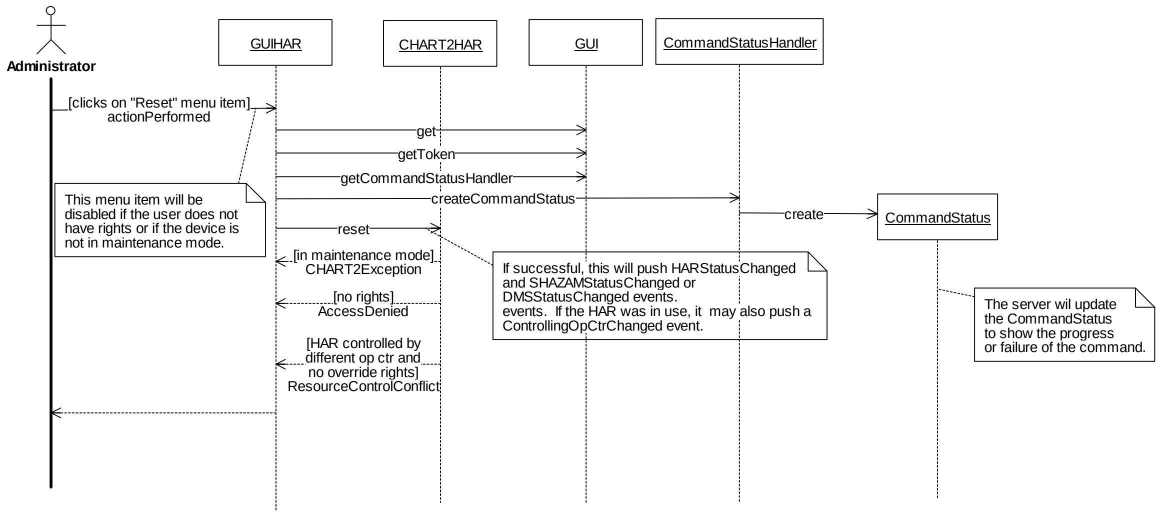


Administrator
GUISHAZAM
SHAZAM
GUI
CommandStatusHandler
[clicks on the "Remove" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
remove
[no rights] accessDenied
This menu item will be disabled if the user does not have the correct rights.
If successful, this will push a SHAZAMRemoved event, which will be caught by the GUI and the GUISHAZAM will be removed from the DataModel as well as removed from any associations with a HAR.
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 86. GUIHARModule:RemoveSHAZAM (Sequence Diagram)

3.8.2.24 GUIHARModule:ResetHAR (Sequence Diagram)

This diagram shows how a HAR is reset in maintenance mode. The Administrator right clicks on a GUIHAR object in the Navigator and clicks on the “Reset” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the reset() method of the CHART2HAR object (which it wraps). If successful, the server will push a CORBA event indicating the changes to the state of the HAR.

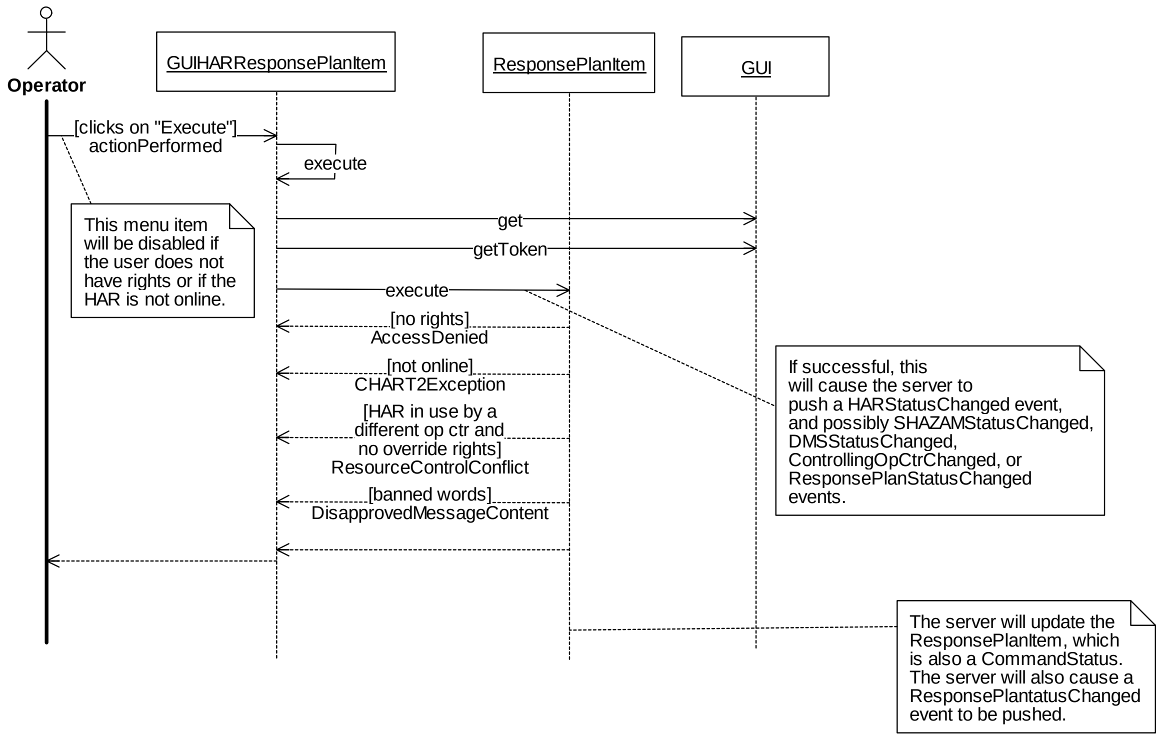


Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Reset" menu item]
actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
reset
This menu item will be disabled if the user does not have rights or if the device is not in maintenance mode.
[in maintenance mode]
CHART2Exception
[no rights]
AccessDenied
[HAR controlled by different op ctr and no override rights]
ResourceControlConflict
If successful, this will push HARStatusChanged and SHAZAMStatusChanged or DMSStatusChanged events. events. If the HAR was in use, it may also push a ControllingOpCtrChanged event.
The server wil update the CommandStatus to show the progress or failure of the command.

Figure 87. GUIHARModule:ResetHAR (Sequence Diagram)

3.8.2.25 GUIHARModule:SetHARMessage (Sequence Diagram)

This diagram shows how a HAR message is set. The Operator right clicks on a GUIHARResponsePlanItem object and clicks on the “Execute” menu item. The GUIHARResponsePlanItem calls the execute() method of the ResponsePlanItem object (which it wraps). If successful, the server will push CORBA events indicating the changes to the state of the HAR. The server will also push events to keep the GUIs updated with the current status of the command.

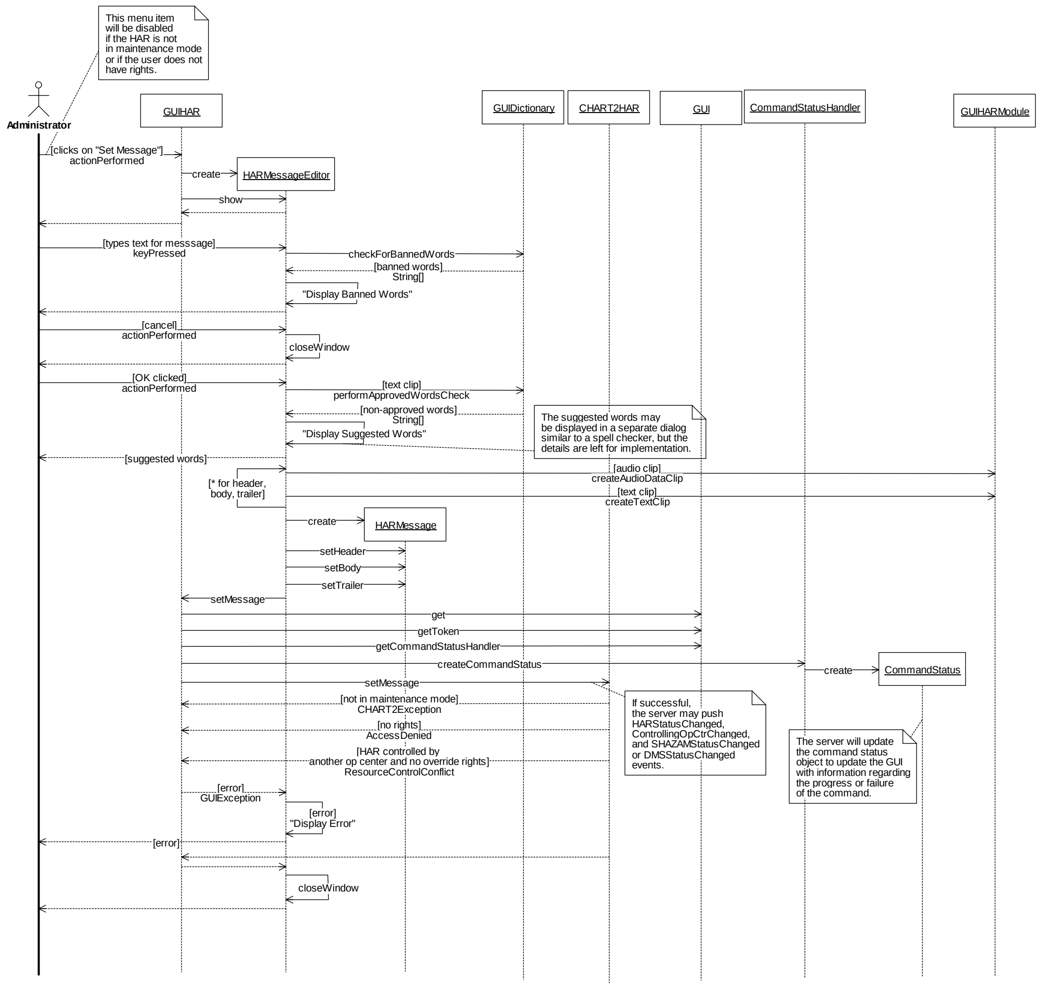


Operator
GUIHARResponsePlanItem
ResponsePlanItem
GUI
[clicks on "Execute"]
actionPerformed
execute
This menu item will be disabled if the user does not have rights or if the HAR is not online.
get
getToken
execute
[no rights]
AccessDenied
[not online]
CHART2Exception
[HAR in use by a different op ctr and no override rights]
ResourceControlConflict
[banned words]
DisapprovedMessageContent
If successful, this will cause the server to push a HARStatusChanged event, and possibly SHAZAMStatusChanged, DMSStatusChanged, ControllingOpCtrChanged, or ResponsePlanStatusChanged events.
The server will update the ResponsePlanItem, which is also a CommandStatus. The server will also cause a ResponsePlantatusChanged event to be pushed.

Figure 88. GUIHARModule:SetHARMessage (Sequence Diagram)

3.8.2.26 GUIHARModule:SetHARMessageInMaintenanceMode (Sequence Diagram)

This shows how a message is set on a HAR when it is in maintenance mode. The user clicks on the GUIHAR object in the Navigator and clicks on the “Set Message” menu item. The GUIHAR object invokes the HARMessageEditor dialog. As the user types a text message, any banned words are displayed if it’s a text message. When the user clicks “OK”, the dialog checks the words (if it’s a text message) and displays any suggestions. If no suggestions are made, the dialog calls the GUIHARModule to create message clips for the header, trailer, and body of the message. The dialog then creates a HARMessage object and inserts the clips into it, then calls the GUIHAR to set the message. The GUIHAR object creates a CommandStatus to monitor the progress of the command, then calls the CHART2HAR object which it wraps. If successful, the server will push CORBA events to update the GUIs for any state changes.

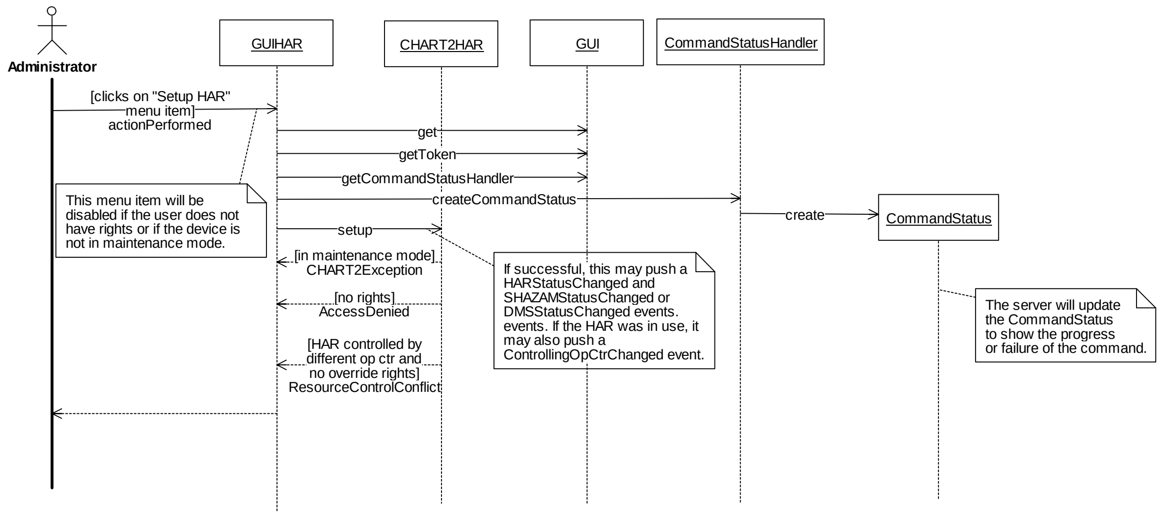


This menu item will be disabled if the HAR is not in maintenance mode or if the user does not have rights.
Administrator
GUIHAR
GUIDictionary
CHART2HAR
GUI
CommandStatusHandler
GUIHARModule
[clicks on "Set Message"] actionPerformed
create
HARMessageEditor
show
[types text for messsage] keyPressed
checkForBannedWords
[banned words] String[]
"Display Banned Words"
[cancel] actionPerformed
closeWindow
[OK clicked] actionPerformed
[text clip] performApprovedWordsCheck
[non-approved words] String[]
"Display Suggested Words"
The suggested words may be displayed in a separate dialog similar to a spell checker, but the details are left for implementation.
[suggested words]
[* for header, body, trailer]
[audio clip] createAudioDataClip
[text clip] createTextClip
create
HARMessage
setHeader
setBody
setTrailer
setMessage
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
setMessage
[not in maintenance mode] CHART2Exception
[no rights] AccessDenied
[HAR controlled by another op center and no override rights] ResourceControlConflict
If successful, the server may push HARStatusChanged, ControllingOpCtrChanged, and SHAZAMStatusChanged or DMSStatusChanged events.
The server will update the command status object to update the GUI with information regarding the progress or failure of the command.
[error] GUIException
[error] "Display Error"
[error]
closeWindow

Figure 89. GUIHARModule:SetHARMessageInMaintenanceMode (Sequence Diagram)

3.8.2.27 GUIHARModule:SetupHAR (Sequence Diagram)

This diagram shows how a HAR is set up in maintenance mode. The Administrator right clicks on a GUIHAR object in the Navigator and clicks on the “Setup” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the setup() method of the CHART2HAR object (which it wraps). If successful, the server will push a CORBA event indicating the changes to the state of the HAR.

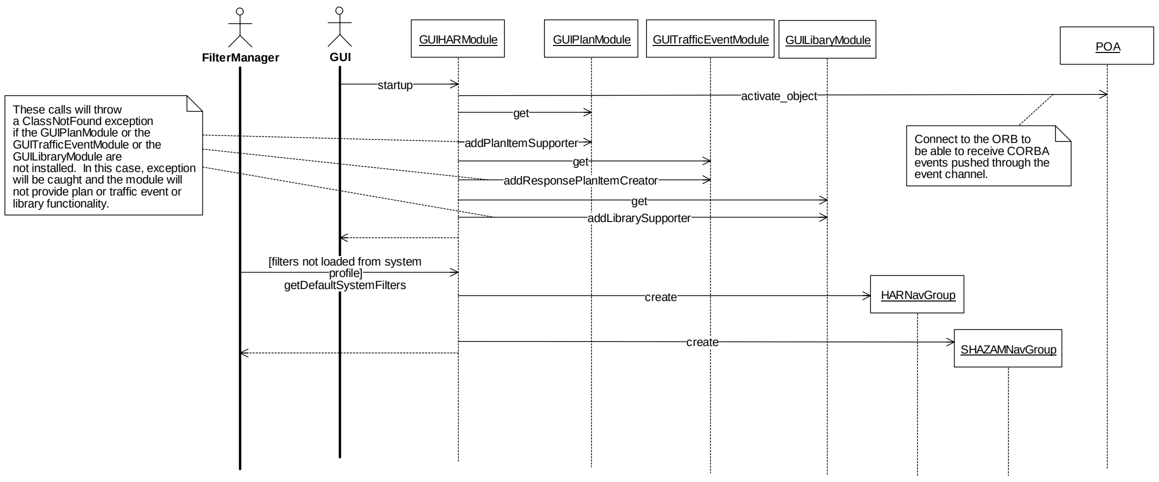


Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Setup HAR" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
This menu item will be disabled if the user does not have rights or if the device is not in maintenance mode.
setup
[in maintenance mode] CHART2Exception
[no rights] AccessDenied
[HAR controlled by different op ctr and no override rights] ResourceControlConflict
If successful, this may push a HARStatusChanged and SHAZAMStatusChanged or DMSStatusChanged events. events. If the HAR was in use, it may also push a ControllingOpCtrChanged event.
The server will update the CommandStatus to show the progress or failure of the command.

Figure 90. GUIHARModule:SetupHAR (Sequence Diagram)

3.8.2.28 Startup:Basic (Sequence Diagram)

This diagram shows the processing that occurs at the GUI's startup. The GUI calls each InstallableModule's startup() method. The GUIHARModule connects itself to the ORB so that it can receive the CORBA events from the Event Service. The GUIHARModule installs itself into the frameworks of the GUIPlanModule, GUILibraryModule, and GUITrafficEventModule so that it can support GUIPlanItem objects, GUIHARStoredMessage objects, and GUIHARResponsePlanItem objects, respectively. It also creates a HARNavGroup that will contain all of the GUIHAR objects in the Navigator, as well as a SHAZAMNavGroup to contain all of the GUISHAZAM objects in the Navigator.



These calls will throw a ClassNotFound exception if the GUIPlanModule or the GUITrafficEventModule or the GUILibraryModule are not installed. In this case, exception will be caught and the module will not provide plan or traffic event or library functionality.

Connect to the ORB to be able to receive CORBA events pushed through the event channel.

Figure 91. Startup:Basic (Sequence Diagram)

3.8.2.29 GUIHARModule:Shutdown (Sequence Diagram)

This diagram shows what happens when the GUI shuts down. The GUI calls all of the InstallableModule objects' shutdown() methods, and the GUIHARModule uses this method to disconnect itself from the ORB.

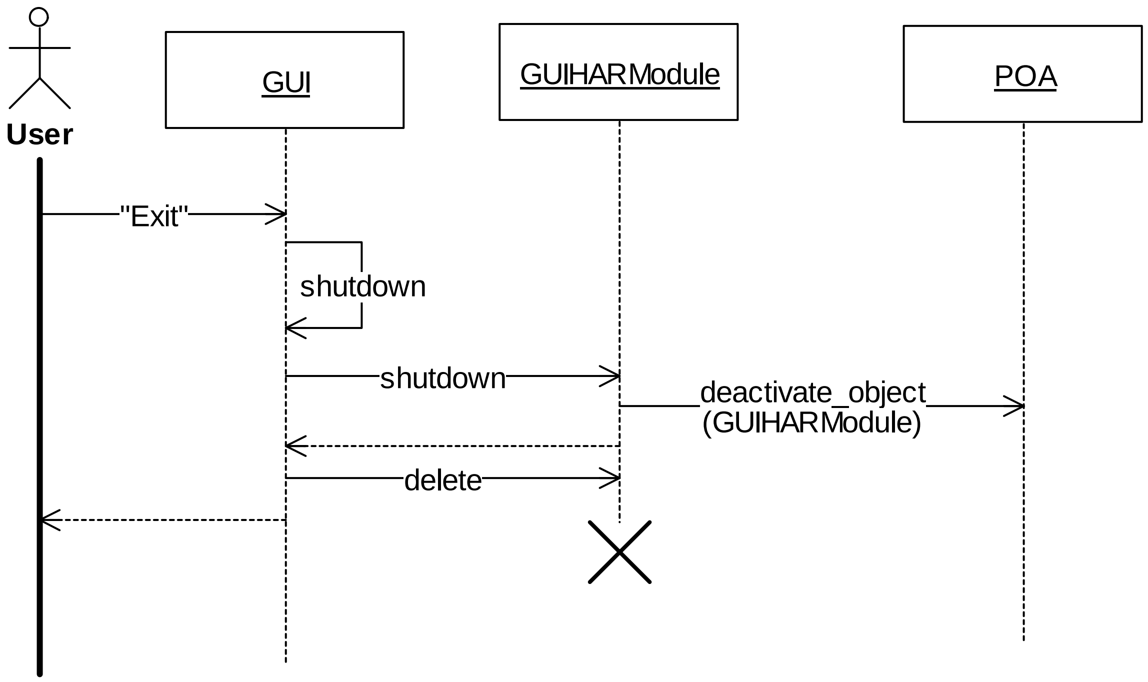


Figure 92. GUIHARModule:Shutdown (Sequence Diagram)

3.8.2.30 GUIHARModule:StoreHARMessageInController (Sequence Diagram)

This diagram shows how a message is stored in a HAR slot in the controller, while the HAR is in maintenance mode. With the HAR Properties Dialog open, the user clicks on the "Store message" button. The dialog calls the GUIHARModule to create a HARMessageClip of the appropriate type (text or voice), depending on the contents of the message, and the GUIHARModule will call the HARFactory to create the clip. The dialog then calls the GUIHAR wrapper object to store the slot message. The GUIHAR object creates a CommandStatus object to monitor the progress of the command, then calls the CHART2HAR to store the slot message. If successful, the server will push a CORBA event update the GUIs with the new state of the HAR.

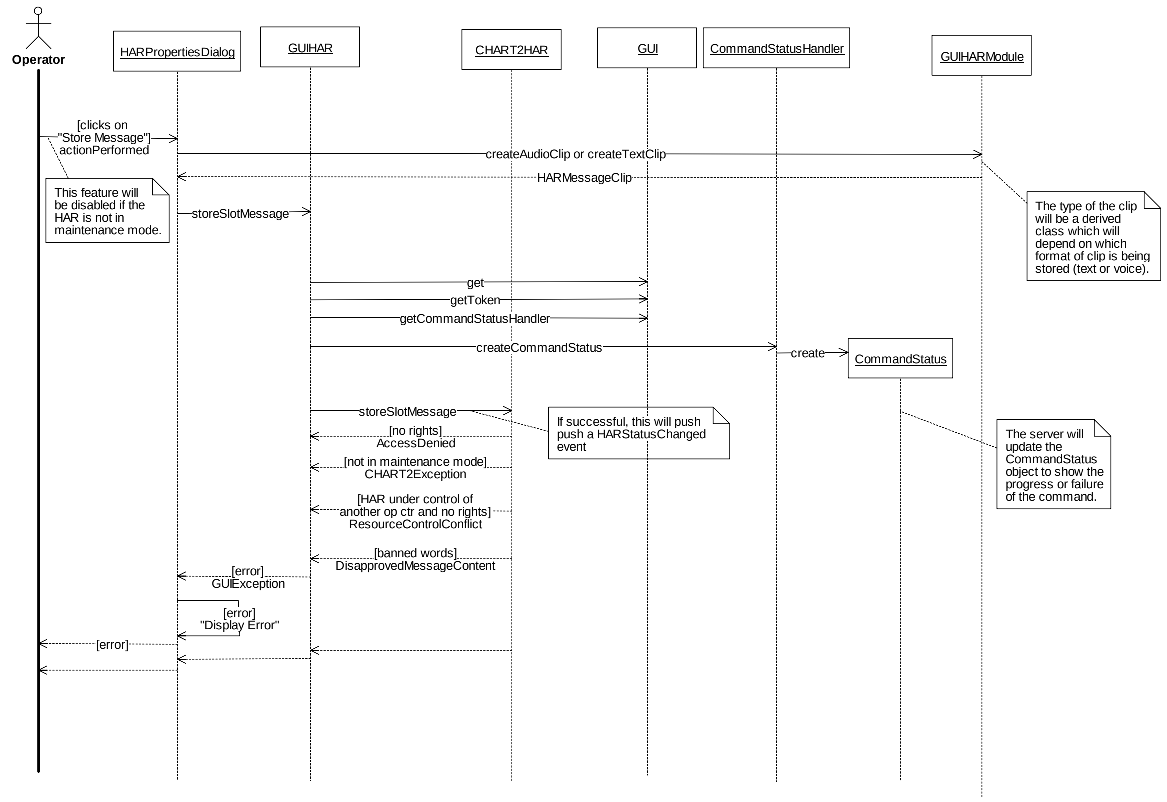
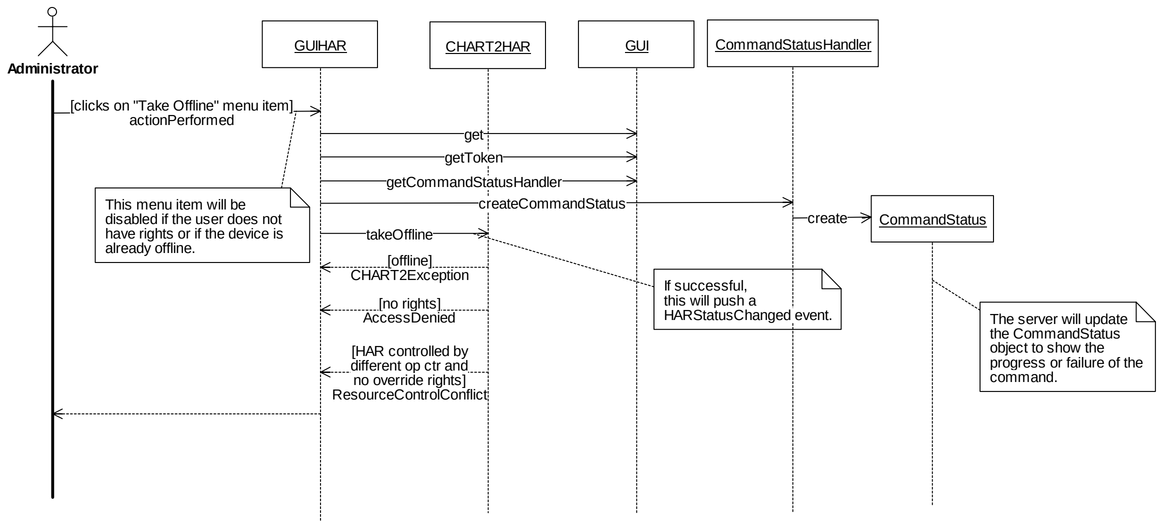

Operator
HARPropertiesDialog
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
GUIHARModule
[clicks on "Store Message"] actionPerformed
createAudioClip or createTextClip
HARMessageClip
This feature will be disabled if the HAR is not in maintenance mode.
storeSlotMessage
The type of the clip will be a derived class which will depend on which format of clip is being stored (text or voice).
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
storeSlotMessage
If successful, this will push push a HARStatusChanged event
[no rights] AccessDenied
[not in maintenance mode] CHART2Exception
[HAR under control of another op ctr and no rights] ResourceControlConflict
The server will update the CommandStatus object to show the progress or failure of the command.
[banned words] DisapprovedMessageContent
[error] GUIException
[error] "Display Error"
[error]

Figure 93. GUIHARModule:StoreHARMessageInController (Sequence Diagram)

3.8.2.31 GUIHARModule:TakeHAROffline (Sequence Diagram)

This diagram shows how a HAR is taken offline. The Administrator right clicks on a GUIHAR object in the Navigator and clicks on the “Take Offline” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the CHART2HAR object (which it wraps) to take it offline. If successful, the server will push a CORBA event indicating that the comm mode has been changed.



Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Take Offline" menu item]
actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
takeOffline
This menu item will be disabled if the user does not have rights or if the device is already offline.
[offline]
CHART2Exception
[no rights]
AccessDenied
[HAR controlled by different op ctr and no override rights]
ResourceControlConflict
If successful, this will push a HARStatusChanged event.
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 94. GUIHARModule:TakeHAROffline (Sequence Diagram)

3.8.2.32 GUIHARModule:TakeSHAZAMOffline (Sequence Diagram)

This diagram shows how a SHAZAM is taken offline. The Administrator right clicks on a GUISHAZAM object in the Navigator and clicks on the “Take Offline” menu item. The GUISHAZAM creates a CommandStatus object to monitor the progress of the command and calls the SHAZAM object (which it wraps) to take it offline. If successful, the server will push a CORBA event indicating that the comm mode has been changed.

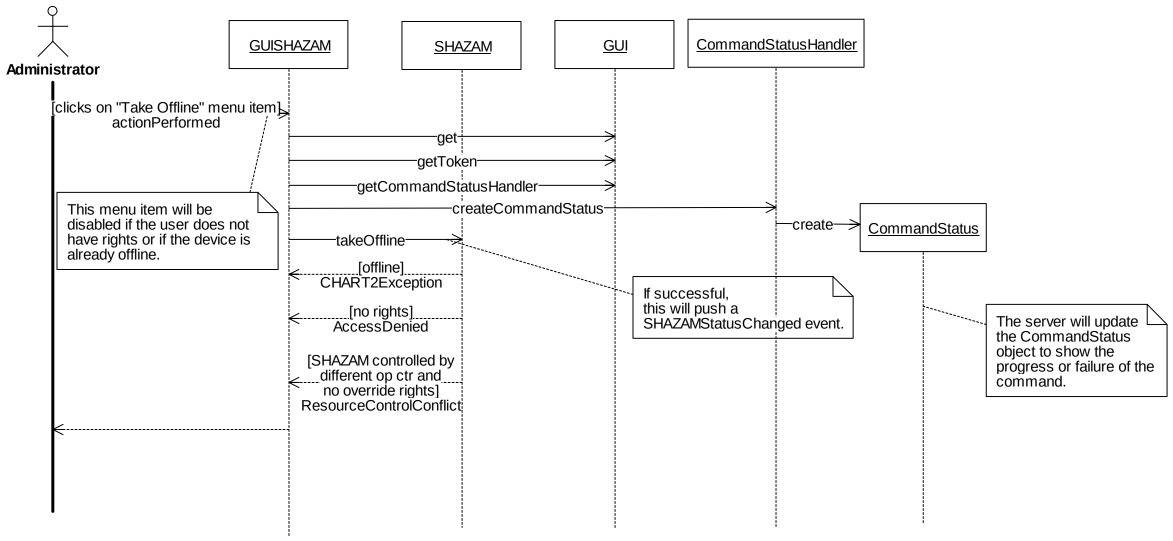

Administrator
GUISHAZAM
SHAZAM
GUI
CommandStatusHandler
[clicks on "Take Offline" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
takeOffline
This menu item will be disabled if the user does not have rights or if the device is already offline.
[offline] CHART2Exception
[no rights] AccessDenied
[SHAZAM controlled by different op ctr and no override rights] ResourceControlConflict
If successful, this will push a SHAZAMStatusChanged event.
The server will update the CommandStatus object to show the progress or failure of the command.

Figure 95. GUIHARModule:TakeSHAZAMOffline (Sequence Diagram)

3.8.2.33 GUIHARModule:TurnOffHARTransmitter (Sequence Diagram)

This diagram shows how a HAR's transmitter is turned off in maintenance mode. The Administrator right clicks on a GUIHAR object in the Navigator and clicks on the "Turn Off Transmitter" menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the setTransmitterOff() method of the CHART2HAR object (which it wraps). If successful, the server will push a CORBA event indicating the changes to the state of the HAR.

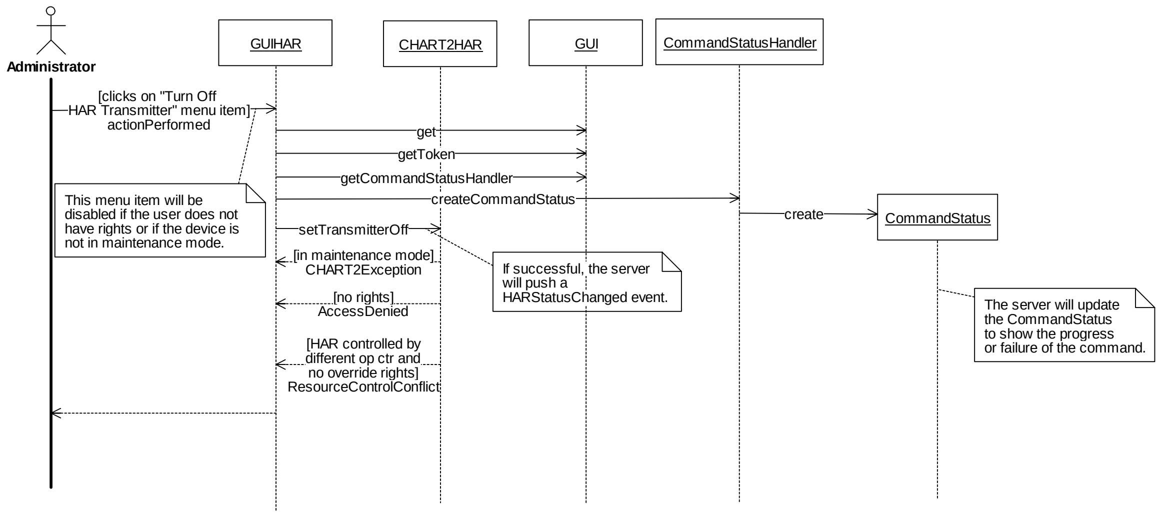

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Turn Off HAR Transmitter" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
setTransmitterOff
This menu item will be disabled if the user does not have rights or if the device is not in maintenance mode.
[in maintenance mode] CHART2Exception
[no rights] AccessDenied
[HAR controlled by different op ctr and no override rights] ResourceControlConflict
If successful, the server will push a HARStatusChanged event.
The server will update the CommandStatus to show the progress or failure of the command.

Figure 96. GUIHARModule:TurnOffHARTransmitter (Sequence Diagram)

3.8.2.34 GUIHARModule:TurnOnHARTransmitter (Sequence Diagram)

This diagram shows how a HAR’s transmitter is turned on in maintenance mode. The Administrator right clicks on a GUIHAR object in the Navigator and clicks on the “Turn On Transmitter” menu item. The GUIHAR creates a CommandStatus object to monitor the progress of the command and calls the setTransmitterOn() method of the CHART2HAR object (which it wraps). If successful, the server will push a CORBA event indicating the changes to the state of the HAR.

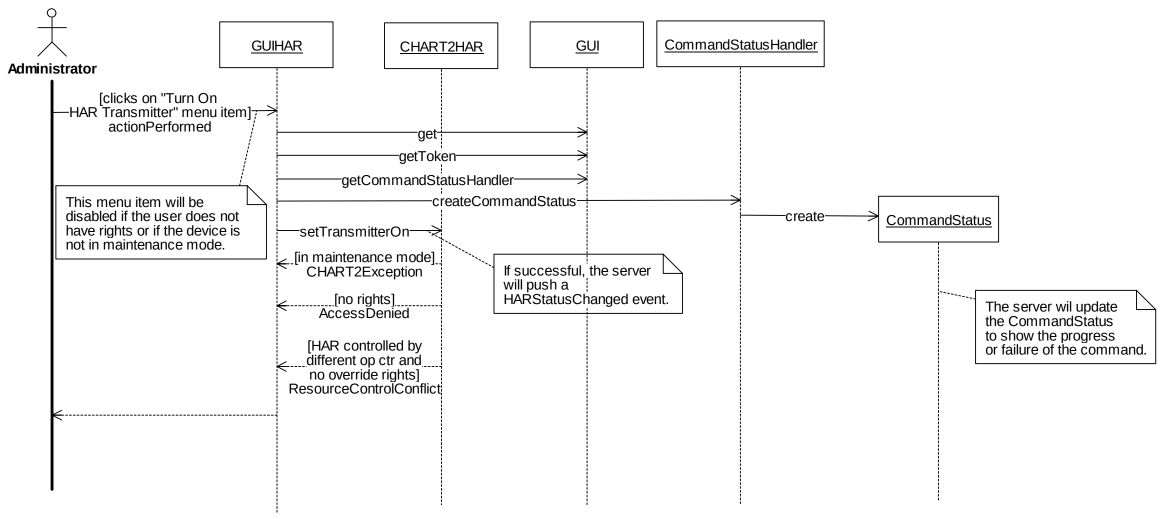

Administrator
GUIHAR
CHART2HAR
GUI
CommandStatusHandler
[clicks on "Turn On HAR Transmitter" menu item] actionPerformed
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
setTransmitterOn
This menu item will be disabled if the user does not have rights or if the device is not in maintenance mode.
[in maintenance mode] CHART2Exception
[no rights] AccessDenied
[HAR controlled by different op ctr and no override rights] ResourceControlConflict
If successful, the server will push a HARStatusChanged event.
The server wil update the CommandStatus to show the progress or failure of the command.

Figure 97. GUIHARModule:TurnOnHARTransmitter (Sequence Diagram)

3.8.2.35 GUIHARModule:ViewHARSlotUsage (Sequence Diagram)

This diagram shows how the HAR slot usage is viewed. The HARPropertiesDialog will call the GUIHAR object to get the slot usage, which will call the HAR object in the server (which it wraps). The HAR object will create some HARMessageClip objects, one for each slot, and the type of the clip object will depend on whether voice or text is being used. The dialog can then display the contents of the clip for each slot that is in use. If the slot contains a recorded message, a HARAudioClip will be returned which can be used to play the message. (See the ListenToHARMessage diagram for details).

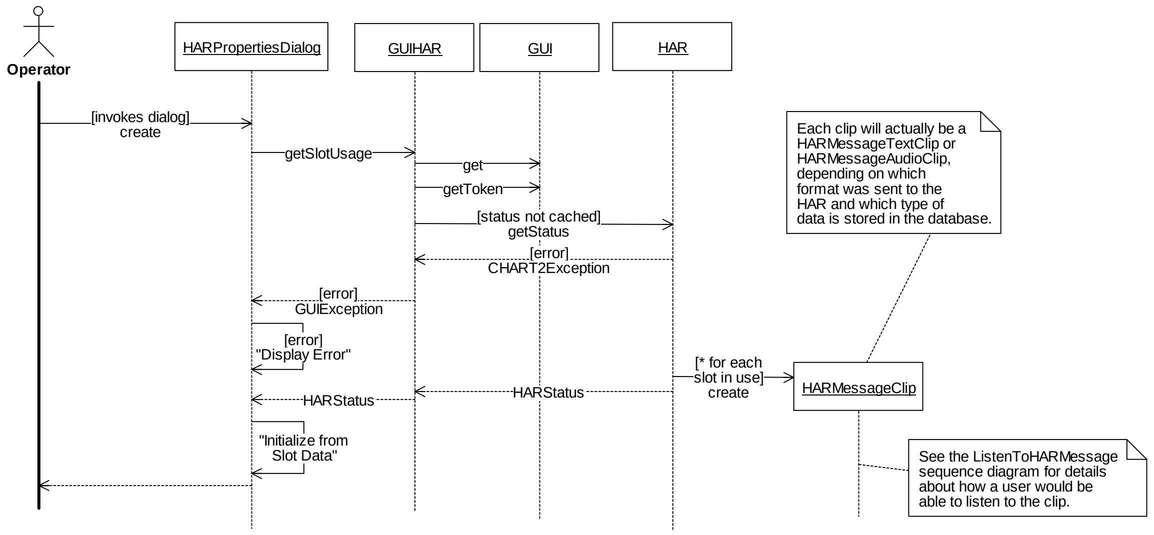

Operator
HARPropertiesDialog
GUIHAR
GUI
HAR
[invokes dialog] create
getSlotUsage
get
getToken
[status not cached] getStatus
[error] CHART2Exception
[error] GUIException
[error] "Display Error"
HARStatus
HARStatus
"Initialize from Slot Data"
[* for each slot in use] create
HARMessageClip
Each clip will actually be a HARMessageTextClip or HARMessageAudioClip, depending on which format was sent to the HAR and which type of data is stored in the database.
See the ListenToHARMessage sequence diagram for details about how a user would be able to listen to the clip.

Figure 98. GUIHARModule:ViewHARSlotUsage (Sequence Diagram)

3.8.2.36 GUIHARModule:ViewHARStoredMessage (Sequence Diagram)

This diagram shows how a HAR stored message is viewed. This is a subset of the ModifyHARStoredMessage sequence diagram, so refer to that diagram for details.

See the ModifyHARStoredMessage sequence diagram for details on how the user may view a HAR stored message.

Figure 99. GUIHARModule:ViewHARStoredMessage (Sequence Diagram)

[DCE:197]

3.9 GUIMessageLibraryModule

3.9.1 Class Diagrams

3.9.1.1 GUIMessageLibraryClasses (Class Diagram)

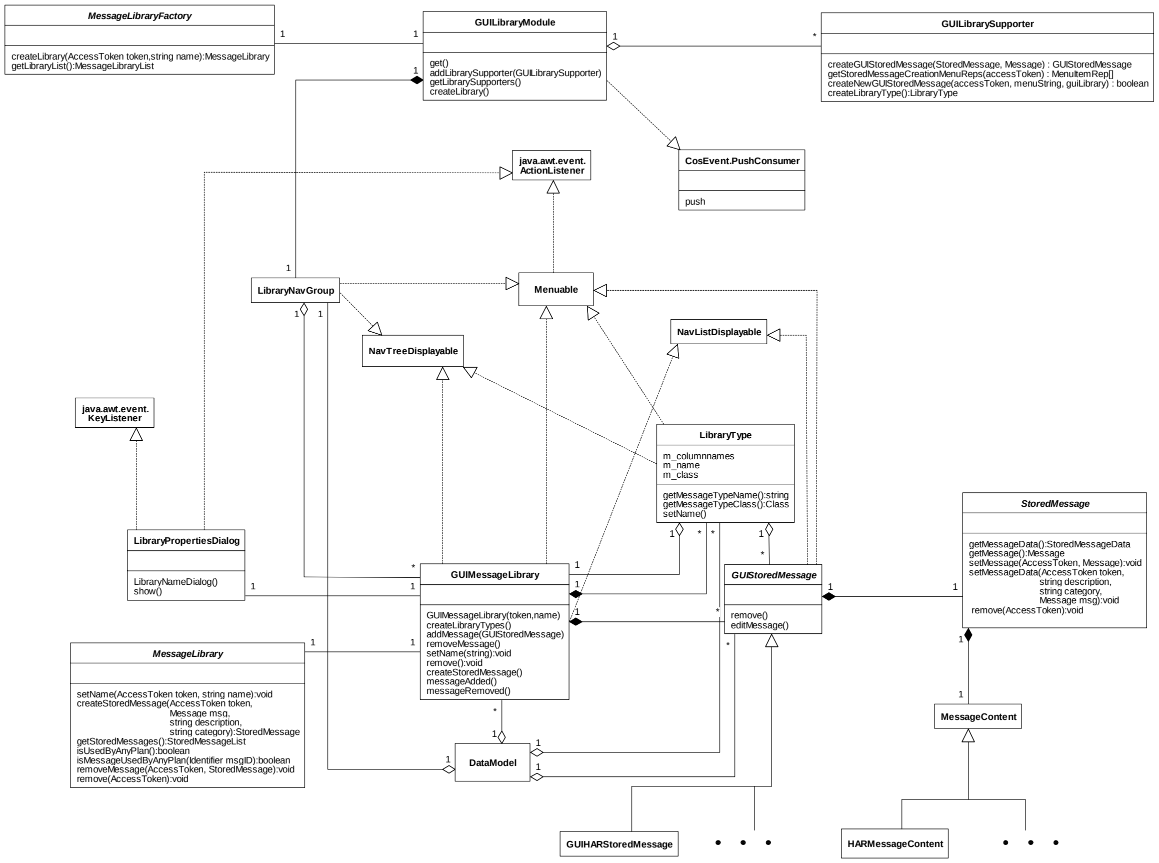

MessageLibraryFactory
createLibrary(AccessToken token,string name):MessageLibrary
getLibraryList():MessageLibraryList
GUILibraryModule
get()
addLibrarySupporter(GUILibrarySupporter)
getLibrarySupporters()
createLibrary()
GUILibrarySupporter
createGUIStoredMessage(StoredMessage, Message) : GUIStoredMessage
getStoredMessageCreationMenuReps(accessToken) : MenuItemRep[]
createNewGUIStoredMessage(accessToken, menuString, guiLibrary) : boolean
createLibraryType():LibraryType
java.awt.event.ActionListener
CosEvent.PushConsumer
push
LibraryNavGroup
Menuable
NavListDisplayable
NavTreeDisplayable
java.awt.event.KeyListener
LibraryType
m_columnnames
m_name
m_class
getMessageTypeName():string
getMessageTypeClass():Class
setName()
StoredMessage
getMessageData():StoredMessageData
getMessage():Message
setMessage(AccessToken, Message):void
setMessageData(AccessToken token, string description, string category, Message msg):void
remove(AccessToken):void
LibraryPropertiesDialog
LibraryNameDialog()
show()
GUIMessageLibrary
GUIMessageLibrary(token,name)
createLibraryTypes()
addMessage(GUIStoredMessage)
removeMessage()
setName(string):void
remove():void
createStoredMessage()
messageAdded()
messageRemoved()
GUIStoredMessage
remove()
editMessage()
MessageLibrary
setName(AccessToken token, string name):void
createStoredMessage(AccessToken token, Message msg, string description, string category):StoredMessage
getStoredMessages():StoredMessageList
isUsedByAnyPlan():boolean
isMessageUsedByAnyPlan(Identifier msgID):boolean
removeMessage(AccessToken, StoredMessage):void
remove(AccessToken):void
DataModel
MessageContent
GUIHARStoredMessage
HARMessageContent
1
*

Figure 100. GUIMessageLibraryClasses (Class Diagram)

3.9.1.1.1 CosEvent.PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.9.1.1.2 GUIHARStoredMessage (Class)

This class provides a GUI "wrapper" object that is used to wrap a StoredMessage CORBA interface that contains HAR-specific data and to supply GUI-specific functionality.

3.9.1.1.3 GUILibrarySupporter (Class)

This class allows the GUILibraryModule to maintain stored messages that have differing formats. When an object of this type is installed the user can create, maintain, and use the specific type of libraries and stored messages that the object supports.

3.9.1.1.4 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.9.1.1.5 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

3.9.1.1.6 GUIMessageLibrary (Class)

This class is a GUI "wrapper" object that is used to wrap a MessageLibrary object. The wrapping is done to cache the data locally for faster access, as well as to give the MessageLibrary some GUI-specific functionality such as menus and command handling.

3.9.1.1.7 GUILibraryModule (Class)

The GUILibraryModule is an installable module in the GUI, and provides all functionality specific to stored message libraries and messages. Only one GUILibraryModule object may exist within the GUI. This class provides the functionality needed to support stored messages and stored message libraries.

3.9.1.1.8 GUIStoredMessage (Class)

This class is a GUI “wrapper” object that is used to wrap a StoredMessage object. It provides a user interface object which can implement whatever interfaces are necessary for the object to exist within the GUI framework (for example, an object must support the NavTreeDisplayable and/or NavListDisplayable interface to be displayed in the Navigator).

3.9.1.1.9 LibraryPropertiesDialog (Class)

This dialog is used to view and edit the stored message library’s name and other properties.

3.9.1.1.10 MessageLibrary (Class)

This class represents a logical collection of messages that are stored in the database.

3.9.1.1.11 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.9.1.1.12 HARMessageContent (Class)

This class represents a HAR message. It consists of header, body and footer of the message that can either be in audio format or plain text.

3.9.1.1.13 java.awt.event. KeyListener (Class)

Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

3.9.1.1.14 LibraryNavGroup (Class)

This class has one instance in the GUILibraryModule. It serves as a container for all of the GUILibrary objects in the module when they are displayed in the Navigator.

3.9.1.1.15 LibraryType (Class)

This object stores information pertaining to each type of stored message library that is supported within the system. It is needed to display different types of messages that have different attributes.

3.9.1.1.16 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.9.1.1.17 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

3.9.1.1.18 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description which are used to allow the user to organize messages.

3.9.1.1.19 MessageContent (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.9.1.1.20 MessageLibraryFactory (Class)

This class is used to create new message libraries and maintain them in a collection.

3.9.2 Sequence Diagrams

3.9.2.1 GUILibraryModule:CreateLibrary (Sequence Diagram)

This diagram shows how a stored message library is created. First, the user right clicks on the Message Library in the navigator and selects Add Library on the menu. This calls the GUILibraryModule that checks the functional rights of the user and will, if the user has the correct rights, display the Properties dialog. The user enters information about the new library and presses enter. The GUILibraryModule is called to create a library. For each different type of stored message supported in the system, the GUILibraryModule creates a LibraryType object that allows the system to properly display message types with different attributes.

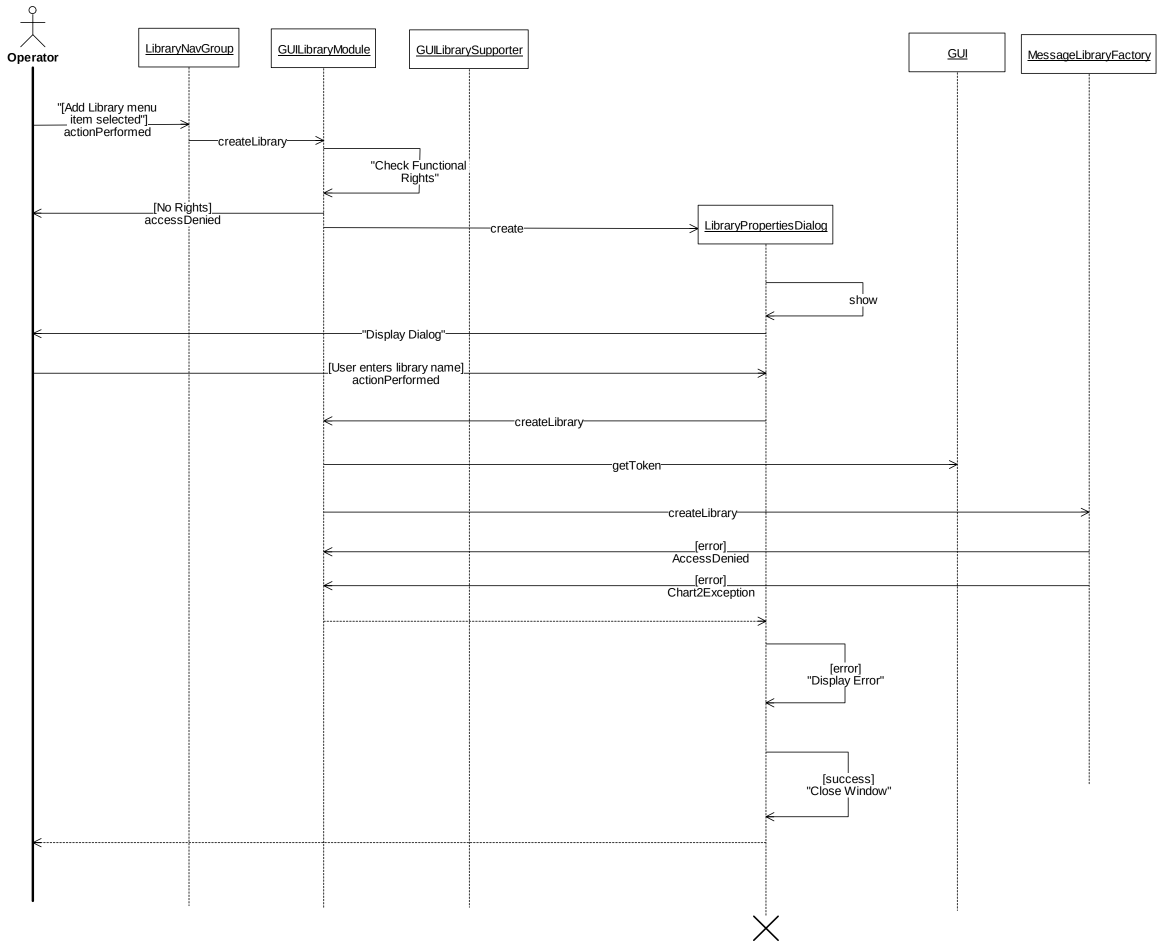
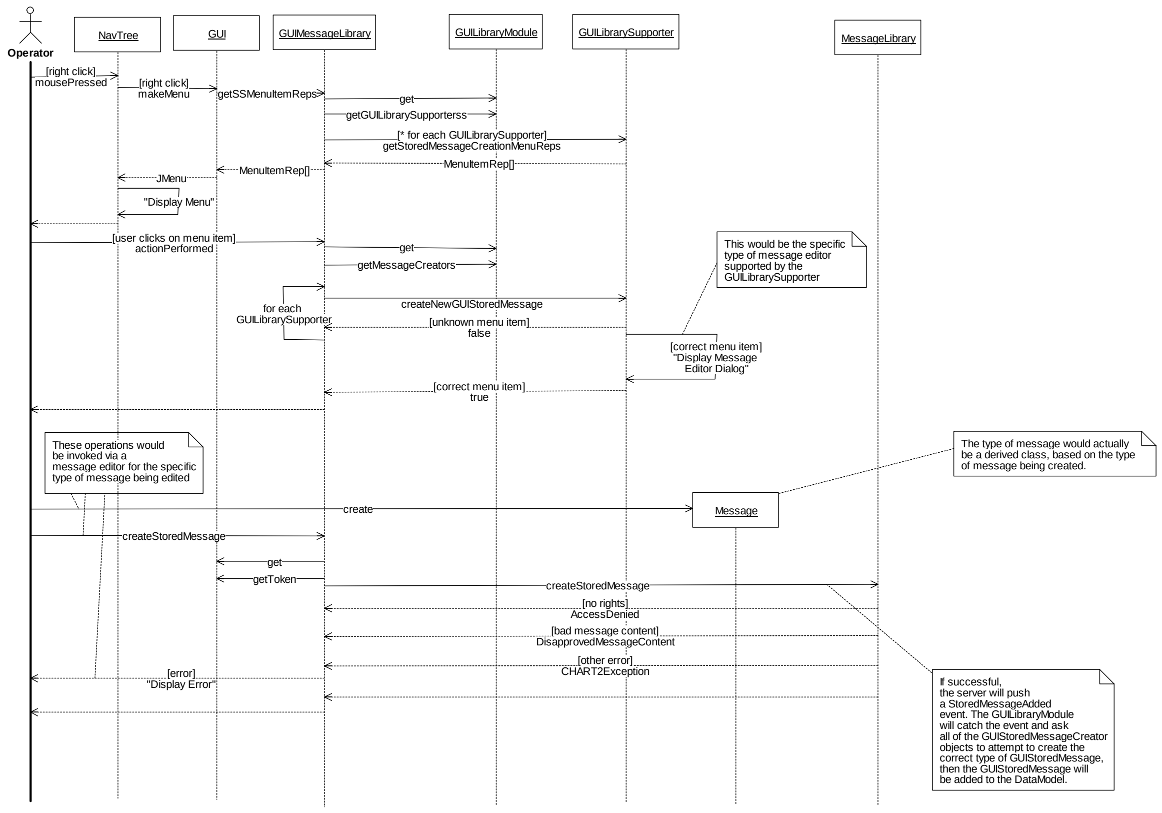


Figure 101. GUILibraryModule:CreateLibrary (Sequence Diagram)

3.9.2.2 GUILibraryModule:CreateStoredMessage (Sequence Diagram)

This diagram shows how a stored message is created. First, the user right clicks on the GUIMessageLibrary object, which calls the GUILibraryModule to get the installed library supporters. Each library supporter returns menu items for message types that it can create. When the user clicks on the appropriate message type, the GUIMessageLibrary object is called again, and this time it asks each library supporter to create the correct type of message based on the menu item. When the correct creator is found it opens its message editor. The operator enters the stored message data. When the data is saved, the information is stored in the database and the server will push a StoredMessageAdded event. The GUILibrary moduleGUILibraryModule then calls the GUIStoredMessage



Operator
NavTree
GUI
GUIMessageLibrary
GUILibraryModule
GUILibrarySupporter
MessageLibrary
[right click] mousePressed
[right click] makeMenu
getSSMenuItemReps
get
getGUILibrarySupporterss
[* for each GUILibrarySupporter] getStoredMessageCreationMenuReps
MenuItemRep[]
MenuItemRep[]
JMenu
"Display Menu"
[user clicks on menu item] actionPerformed
get
getMessageCreators
This would be the specific type of message editor supported by the GUILibrarySupporter
for each GUILibrarySupporter
createNewGUIStoredMessage
[unknown menu item] false
[correct menu item] "Display Message Editor Dialog"
[correct menu item] true
These operations would be invoked via a message editor for the specific type of message being edited
The type of message would actually be a derived class, based on the type of message being created.
create
Message
createStoredMessage
get
getToken
createStoredMessage
[no rights] AccessDenied
[bad message content] DisapprovedMessageContent
[other error] CHART2Exception
[error] "Display Error"
If successful, the server will push a StoredMessageAdded event. The GUILibraryModule will catch the event and ask all of the GUIStoredMessageCreator objects to attempt to create the correct type of GUIStoredMessage, then the GUIStoredMessage will be added to the DataModel.

Figure 102. GUILibraryModule:CreateStoredMessage (Sequence Diagram)

3.9.2.3 GUILibraryModule:DeleteLibrary (Sequence Diagram)

This diagram shows how a stored message library is removed from the system. First, the user right clicks on the Message Library in the navigator and selects Delete Library on the menu. This calls the MessageLibrary object, which removes the library and any stored messages, contained in the library.

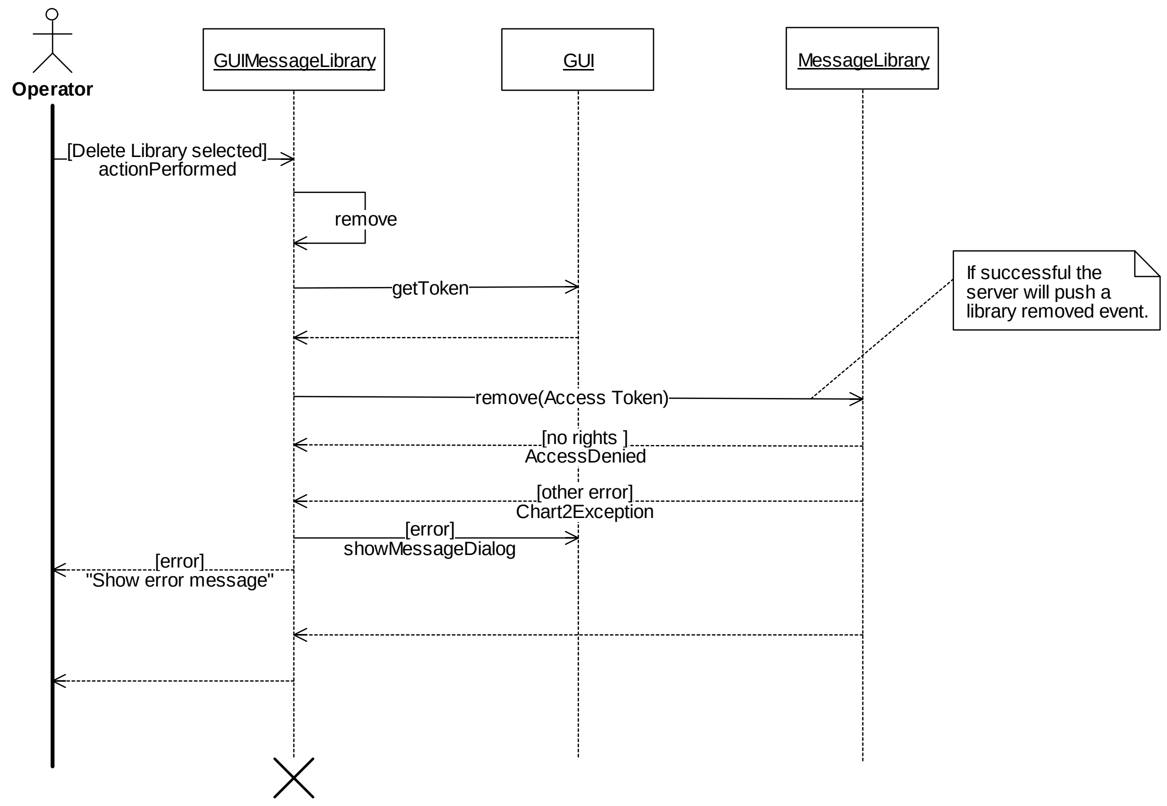

Operator
GUIMessageLibrary
GUI
MessageLibrary
[Delete Library selected] actionPerformed
remove
getToken
If successful the server will push a library removed event.
remove(Access Token)
[no rights] AccessDenied
[other error] Chart2Exception
[error] showMessageDialog
[error] "Show error message"

Figure 103. GUILibraryModule:DeleteLibrary (Sequence Diagram)

3.9.2.4 GUILibraryModule:DeleteStoredMessage (Sequence Diagram)

This diagram shows how a stored message is deleted. First, the user right clicks on the GUIMessageLibrary object, which calls the GUIStoredMessage to remove itself from the system. Once the object is removed from the system the HandleEventLibraryRemoved diagram shows how the navigator is updated for all users.

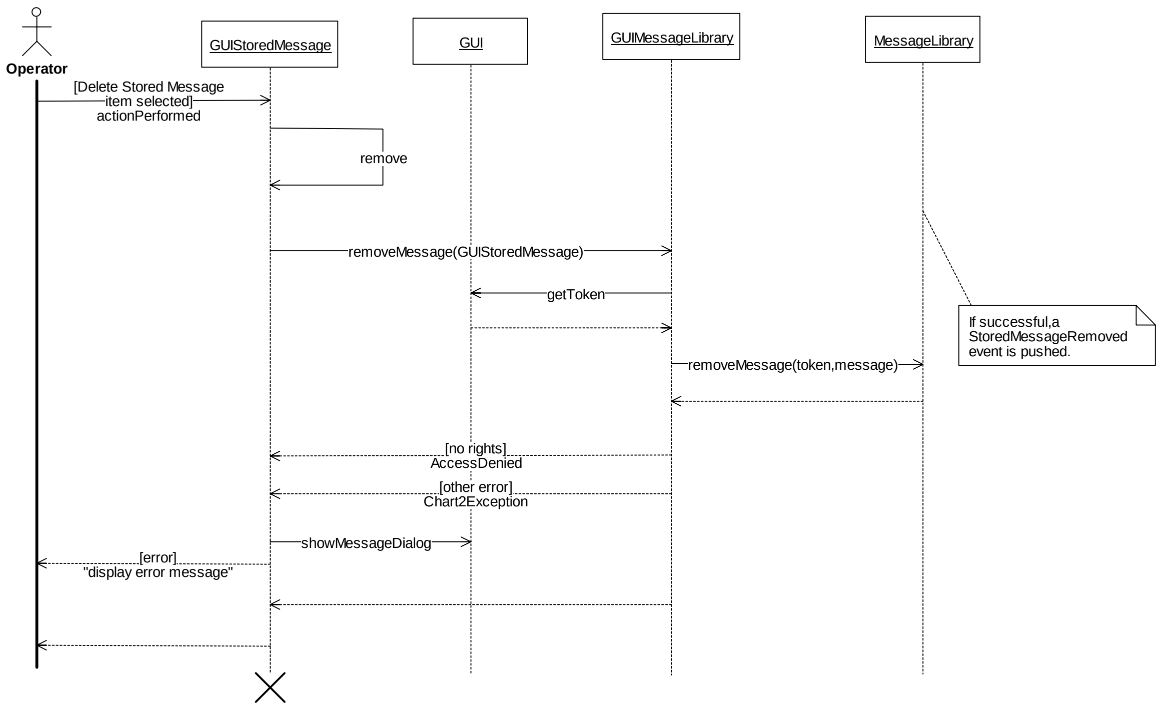
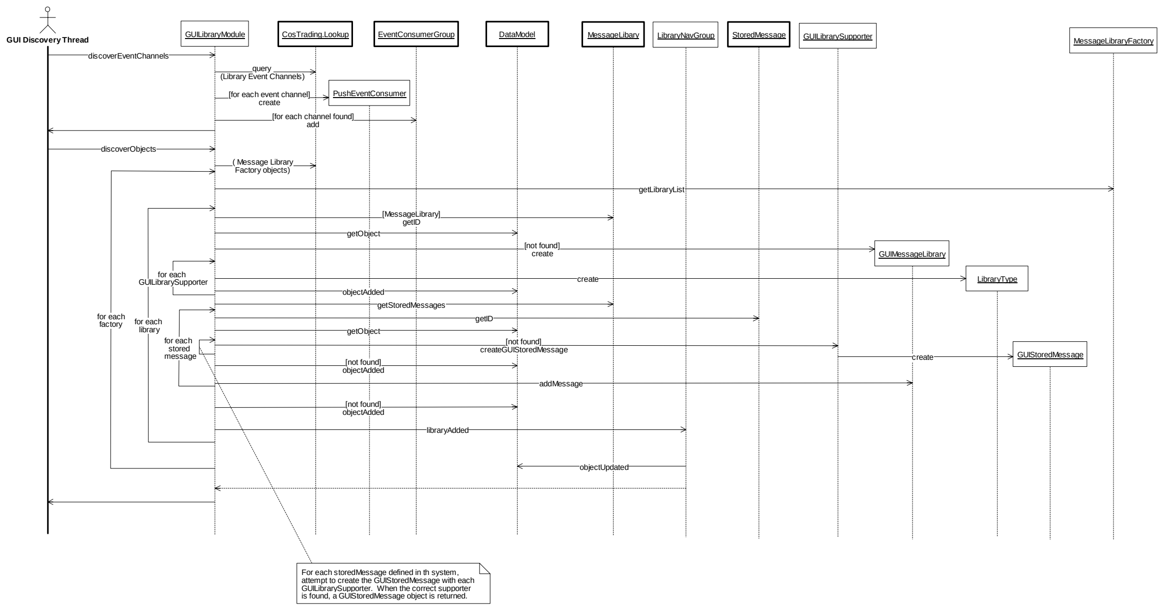

Operator
GUIStoredMessage
GUI
GUIMessageLibrary
MessageLibrary
[Delete Stored Message item selected] actionPerformed
remove
removeMessage(GUIStoredMessage)
getToken
removeMessage(token,message)
If successful,a StoredMessageRemoved event is pushed.
[no rights] AccessDenied
[other error] Chart2Exception
showMessageDialog
[error] "display error message"

Figure 104. GUILibraryModule:DeleteStoredMessage (Sequence Diagram)

3.9.2.5 GUILibraryModule:Discovery (Sequence Diagram)

This diagram shows how the Library Module event channels and the library and stored message objects are discovered and added to the system. This will be a periodic process, and the GUI will call the GUILibraryModulereeatedly. When the GUI asks the module to discover event channels, it looks up the library event channels in the trader. It then creates a PushEventConsumer and adds it to the EventConsumerGroup, which actually attaches the consumer to the channel and reattches it if the event service is restarted. (Duplicate channels will be ignored). The GUI then calls the module to dicover objects. At this time the module will query the Library objects in the trader. If any are found it will create an Identifier to be used as a lookup key for use with the DataModel. For each library found and added to the DataModel the module finds all stored messages. To create a GUIStoredMessage wrapper object, the module attemts to create the stored message using each installed GUILibrarySupporter. When the correct supporter is used the wrapper object is created and added to the DataModel and the GUIMessageLibrary objects.



For each storedMessage defined in th system, attempt to create the GUIStoredMessage with each GUILibrarySupporter. When the correct supporter is found, a GUIStoredMessage object is returned.

Figure 105. GUILibraryModule:Discovery (Sequence Diagram)

3.9.2.6 GUILibraryModule:HandleEventLibraryAdded (Sequence Diagram)

This diagram shows how the GUI receives information when a library is added to the system from the CORBA Event service and displays it to the user once the DataModel is updated.

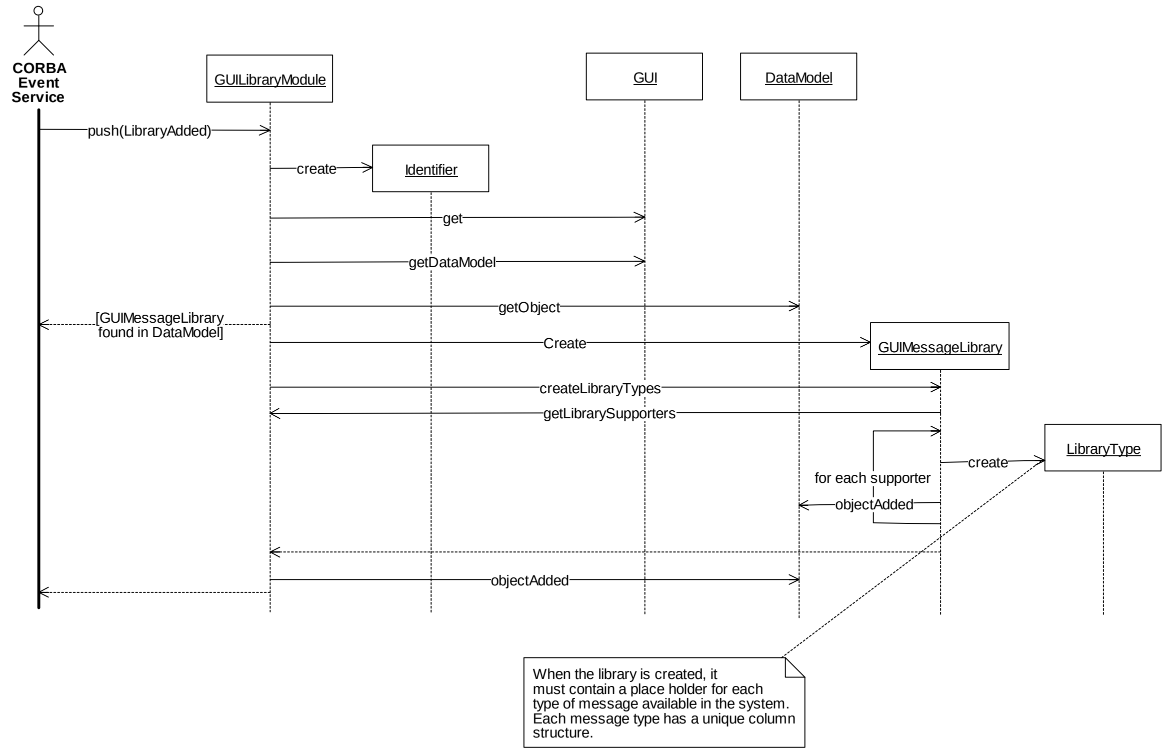


Figure 106. GUILibraryModule:HandleEventLibraryAdded (Sequence Diagram)

3.9.2.7 GUILibraryModule:HandleEventLibraryNameChange (Sequence Diagram)

This diagram shows how the GUI receives information when a library name is changed from the CORBA Event service and displays it to the user once the DataModel is updated.

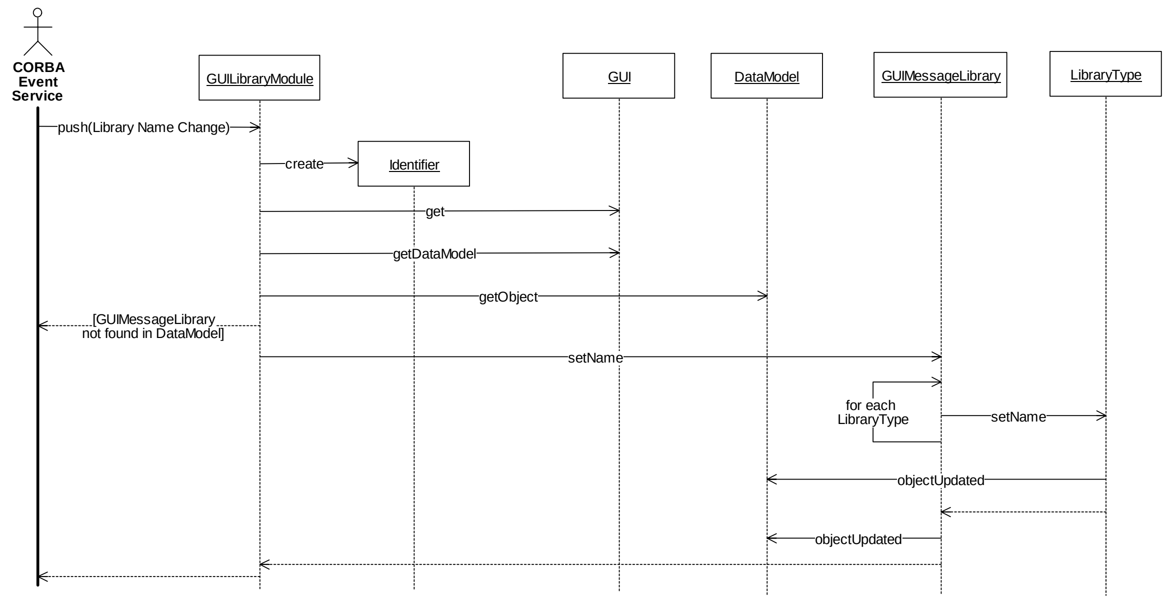


Figure 107. GUILibraryModule:HandleEventLibraryNameChange (Sequence Diagram)

3.9.2.8 GUILibraryModule:HandleEventLibraryRemoved (Sequence Diagram)

This diagram shows how the GUI receives information when a library is removed from the system from the CORBA Event service and displays it to the user once the DataModel is updated.

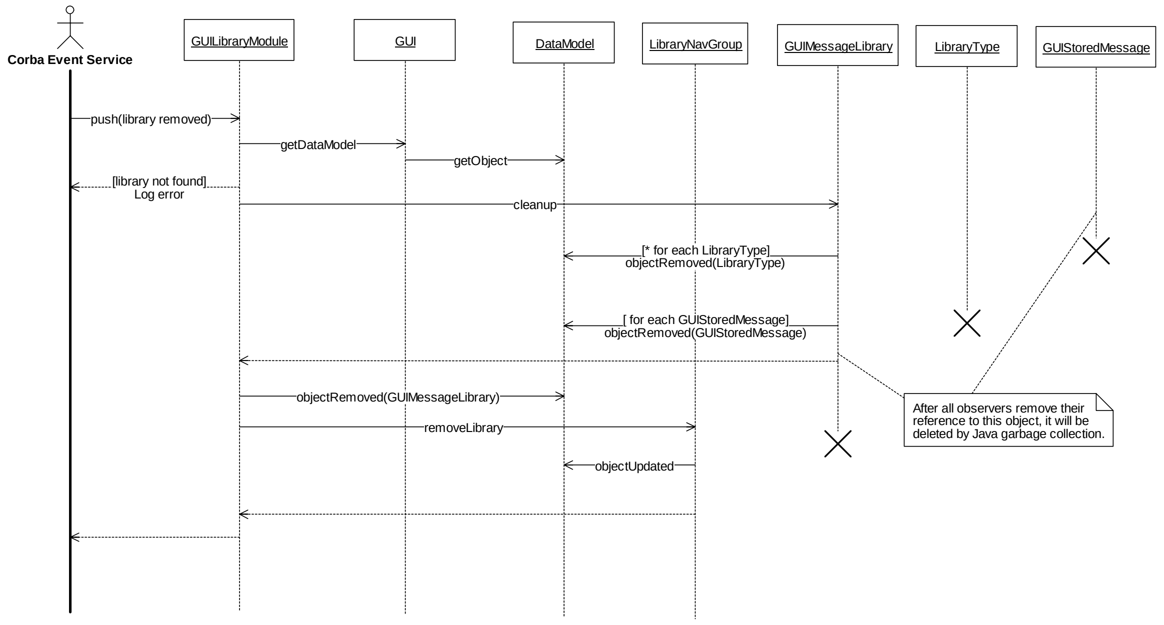
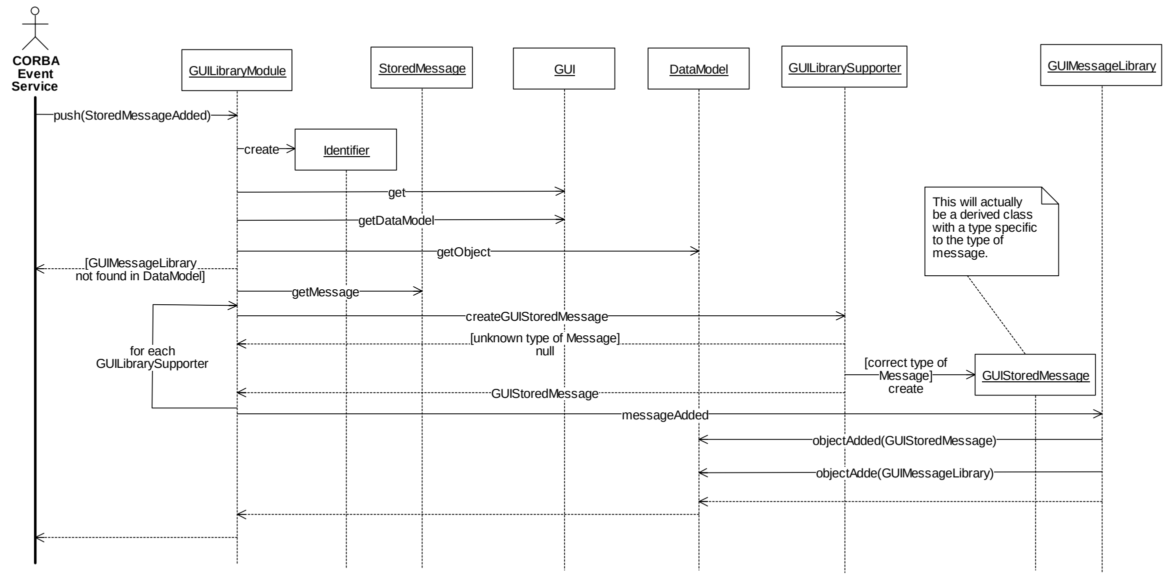


Figure 108. GUILibraryModule:HandleEventLibraryRemoved (Sequence Diagram)

3.9.2.9 GUILibraryModule:HandleEventStoredMessageAdded (Sequence Diagram)

This diagram shows how the GUI receives information when a stored mesage is added to the system from the CORBA Event service and displays it to the user once the DataModel is updated.



CORBA Event Service
GUILibraryModule
StoredMessage
GUI
DataModel
GUILibrarySupporter
GUIMessageLibrary
push(StoredMessageAdded)
create
Identifier
get
getDataModel
getObject
[GUIMessageLibrary not found in DataModel]
getMessage
createGUIStoredMessage
[unknown type of Message] null
for each GUILibrarySupporter
This will actually be a derived class with a type specific to the type of message.
[correct type of Message] create
GUIStoredMessage
GUIStoredMessage
messageAdded
objectAdded(GUIStoredMessage)
objectAdde(GUIMessageLibrary)

Figure 109. GUILibraryModule:HandleEventStoredMessageAdded (Sequence Diagram)

3.9.2.10 GUILibraryModule:HandleEventStoredMessageRemoved (Sequence Diagram)

This diagram shows how the GUI receives information from the CORBA Event service when a stored message is removed from the system, and displays it to the user once the DataModel is updated.

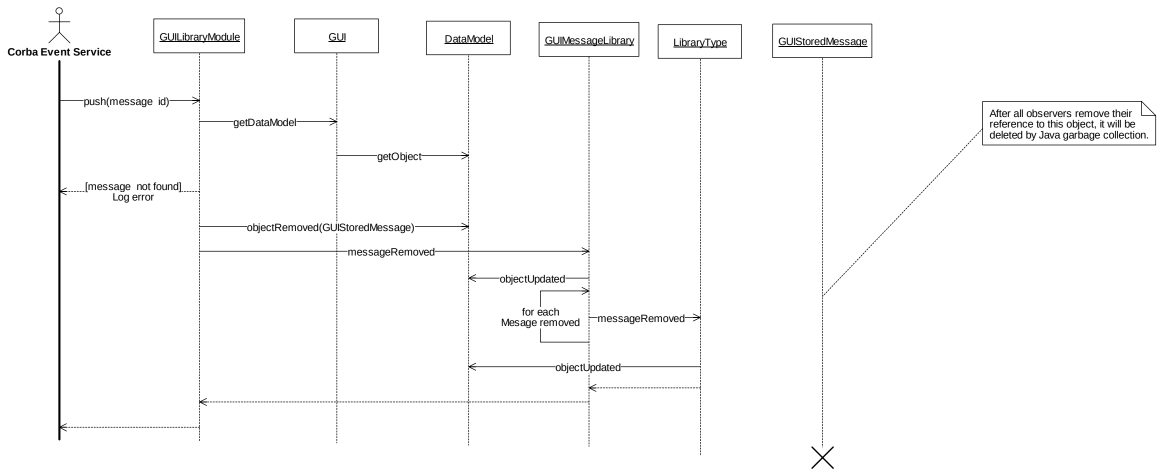


Figure 110. GUILibraryModule:HandleEventStoredMessageRemoved (Sequence Diagram)

3.9.2.11 GUILibraryModule:Login (Sequence Diagram)

This diagram shows what happens when the user logs on.

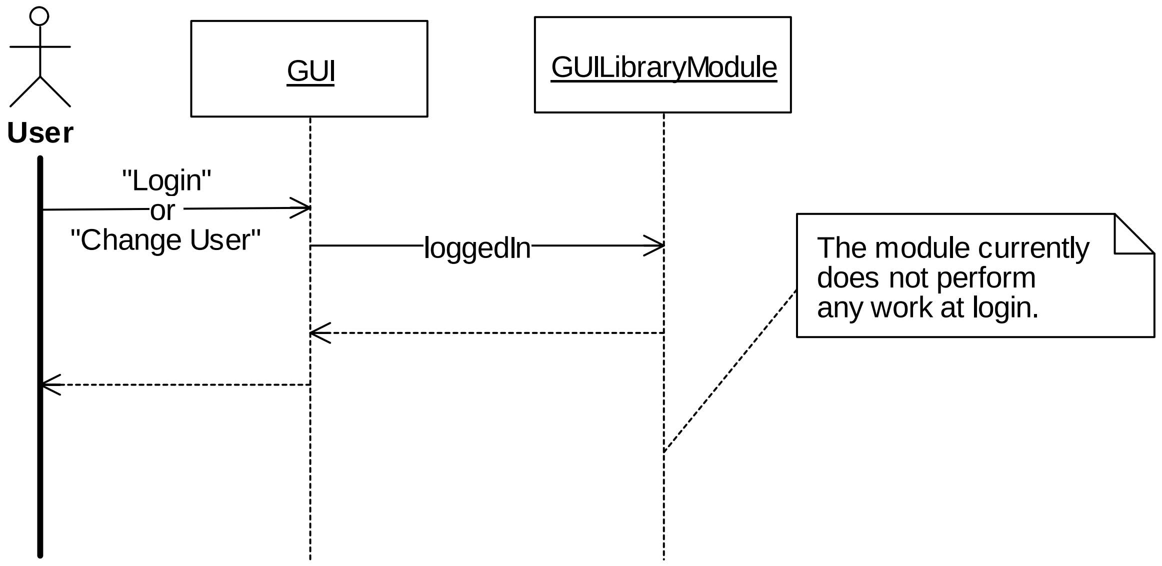


Figure 111. GUILibraryModule:Login (Sequence Diagram)

3.9.2.12 GUILibraryModule:Logout (Sequence Diagram)

This diagram shows what happens when the user logs out of the system.

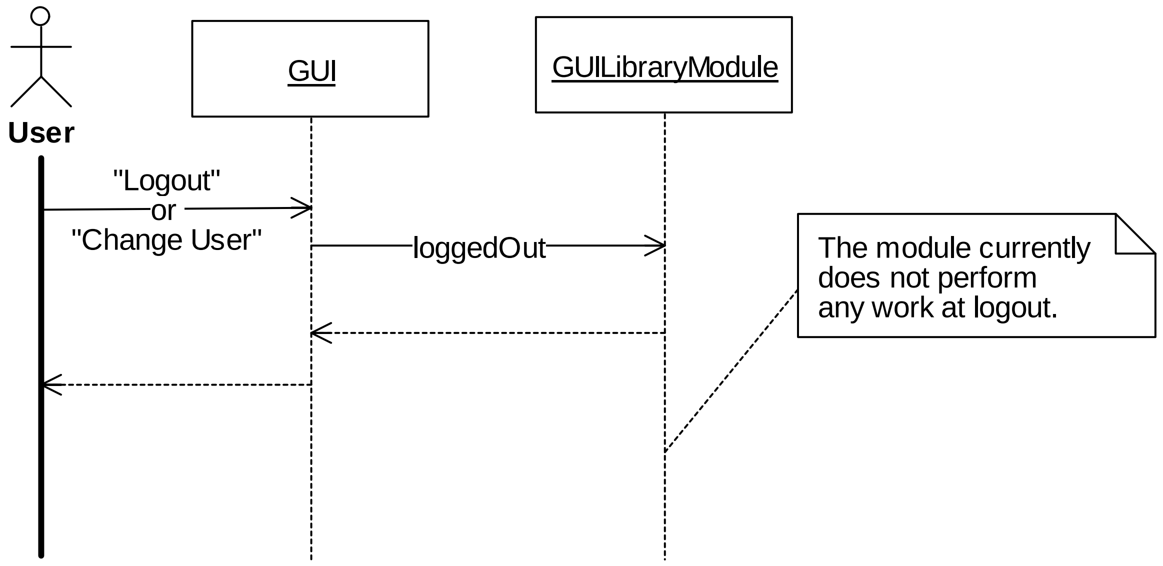


Figure 112. GUILibraryModule:Logout (Sequence Diagram)

3.9.2.13 GUILibraryModule:SetLibraryName (Sequence Diagram)

This diagram shows the steps taken to change the name of an existing library. The user right clicks on a library and selects the Set Name option. A dialog is displayed and the user is allowed to enter a new name. On pressing enter the name is changed.

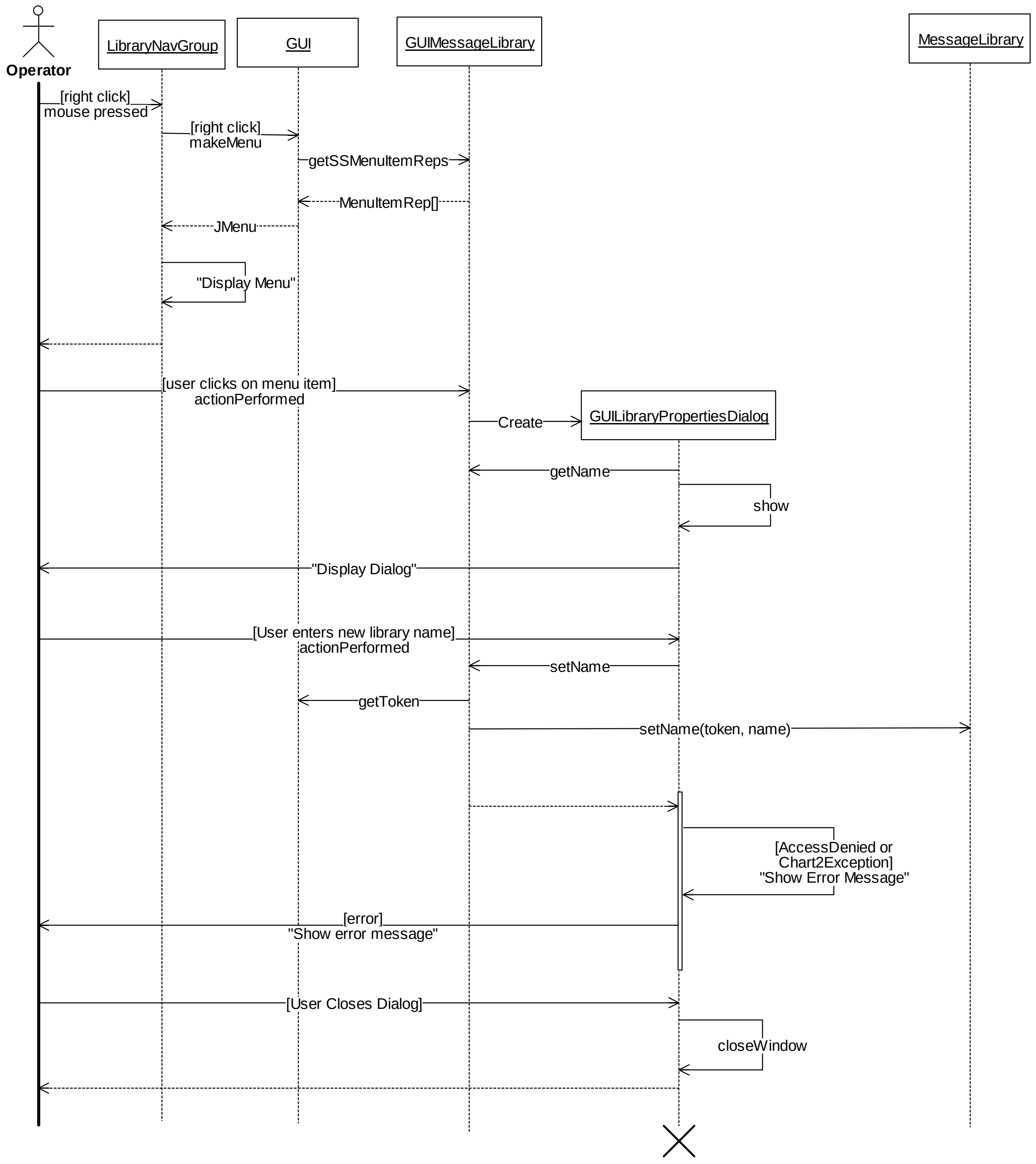


Figure 113. GUILibraryModule:SetLibraryName (Sequence Diagram)

3.9.2.14 GUILibraryModule:Shutdown (Sequence Diagram)

This diagram shows what happens when the GUI shuts down. The GUI cals all of the InstallableModule objects' shutdown() methods, and the GUILibraryModule disconnects from the ORB.

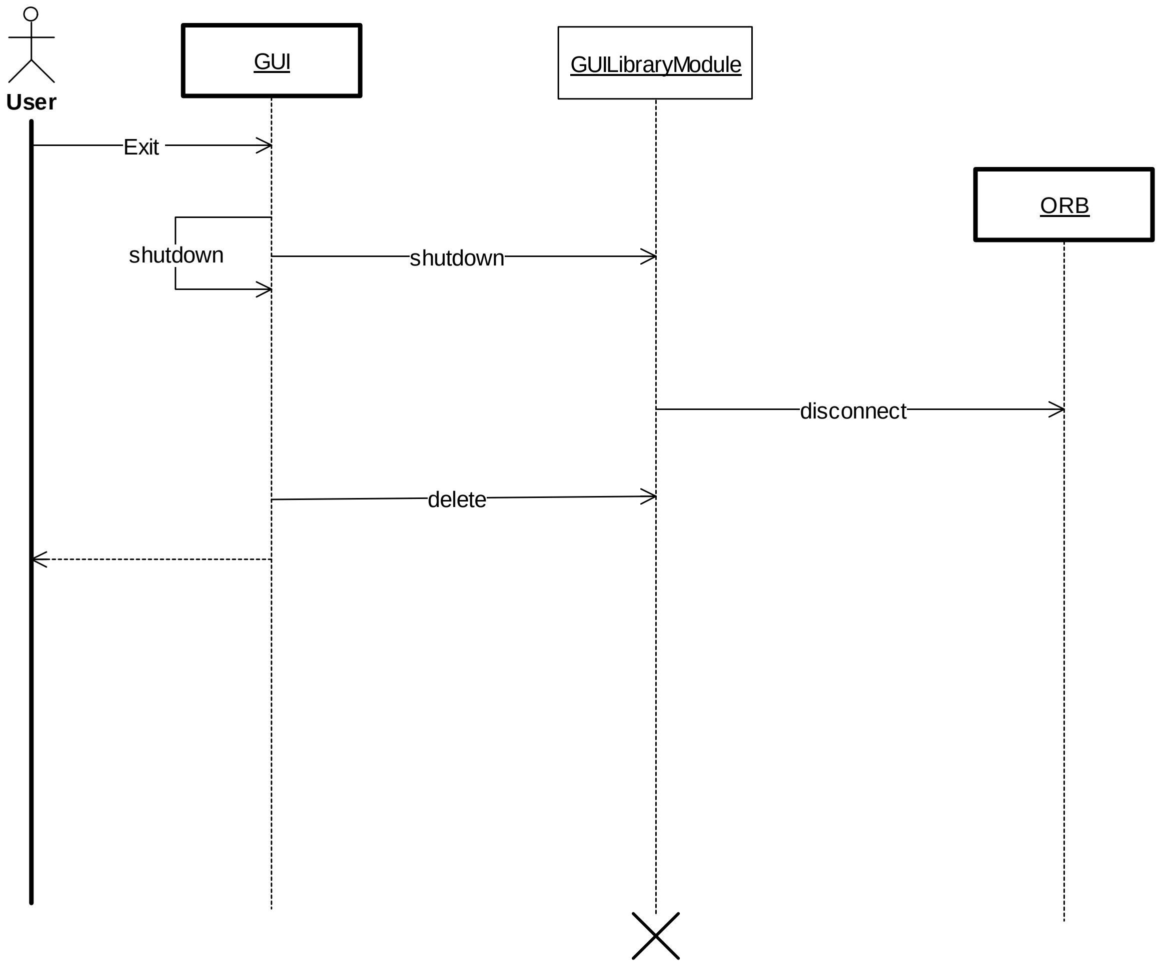


Figure 114. GUILibraryModule:Shutdown (Sequence Diagram)

3.9.2.15 GUILibraryModule:Startup (Sequence Diagram)

This diagram shows the steps taken to initialize the GUILibraryModule. On startup the module creates the LibraryNavGroup for display of he libraries in the navigator and connects to the ORB.

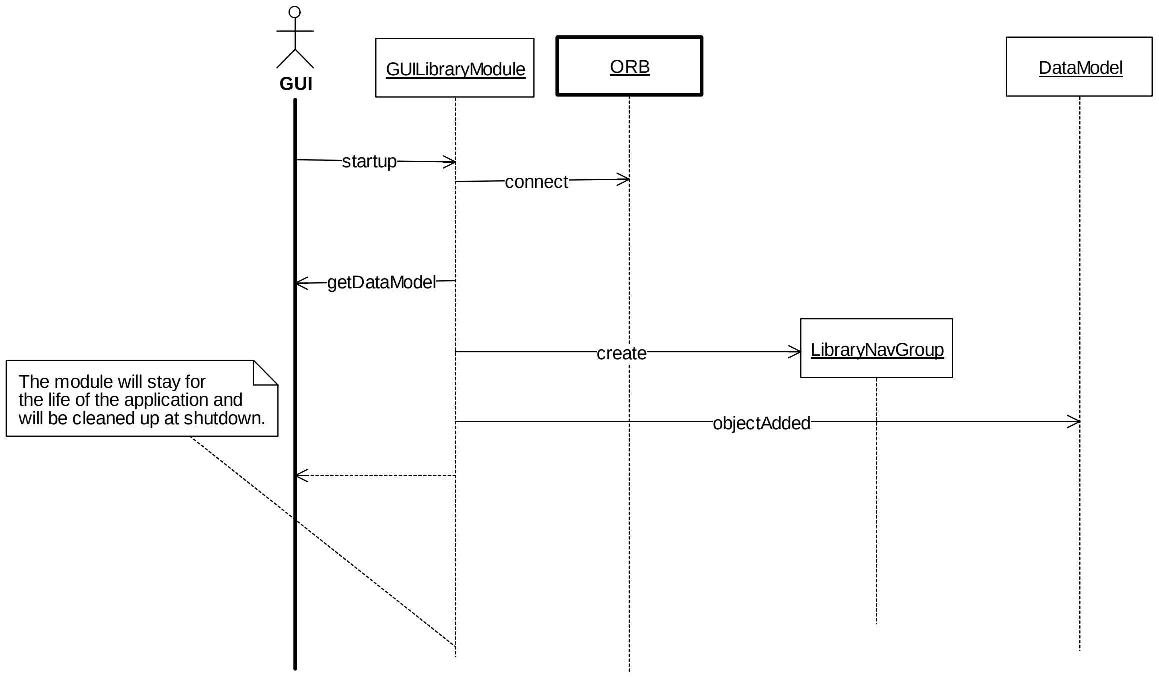


Figure 115. GUILibraryModule:Startup (Sequence Diagram)

3.10 GUIPlanModule

3.10.1 Class Diagrams

3.10.1.1 GUIPlanClasses (Class Diagram)

This diagram shows the classes used by the GUIPlan module and their relationships.

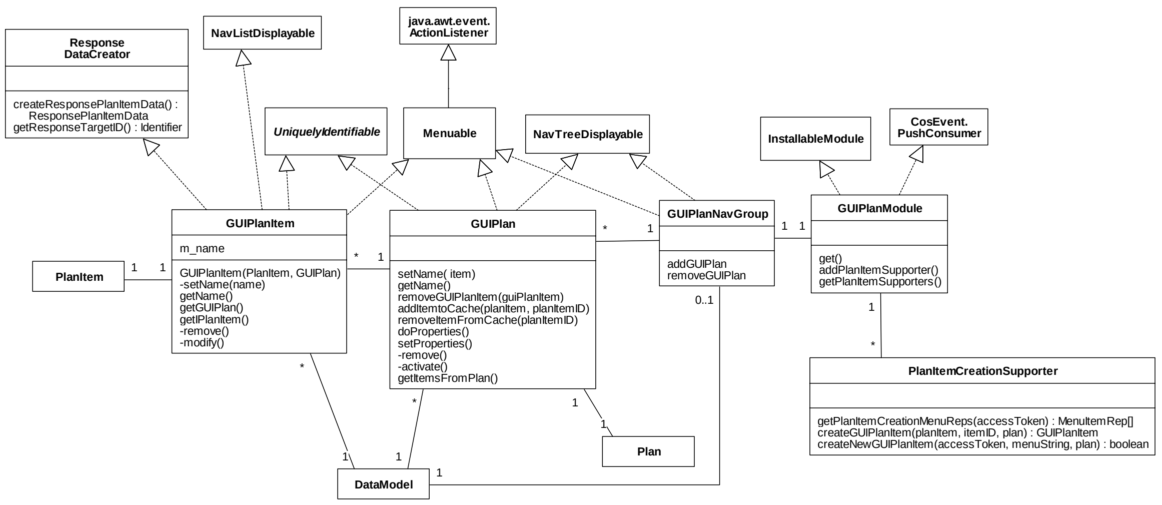


Figure 116. GUIPlanClasses (Class Diagram)

3.10.1.1.1 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.10.1.1.2 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.10.1.1.3 GUIPlan (Class)

This class is a GUI wrapper for the Plan object. The wrapping is done to cache the data locally for faster access, as well as to give the Plan some GUI-specific functionality such as menus and command handling.

3.10.1.1.4 GUIPlanItem (Class)

This is a GUI base class for all the plan items. Each GIUPlanItem object will serve as a GUI wrapper to cache the plan item data locally and also to handle all user interaction in the GUI, such as menus and command handling.

3.10.1.1.5 GUIPlanModule (Class)

This is an installable GUI module that handles the Plan functionality in the GUI. Other modules that support plan items must attach their PlanItemCreationSupporters to the GUIPlanModule at startup. The plan module will call the supporters when it is necessary to create a specific type of GUIPlanItem.

3.10.1.1.6 GUIPlanNavGroup (Class)

This class serves as a container for all of the GUIPlan objects in the GUIPlanModule, when they are displayed in the navigator. It provides functionality for displaying menus. The GUIPlanModule has one instance of this class.

3.10.1.1.7 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.10.1.1.8 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.10.1.1.9 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

3.10.1.1.10 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.10.1.1.11 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

3.10.1.1.12 Plan (Class)

A Plan is a group of actions that are listed out in advance to be used in response to a traffic event. Each action is defined to be a Plan item. The Plan supports functionality to add and remove plan items.

3.10.1.1.13 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This CORBA interface is subclassed for specific actions that can be planned in the system.

3.10.1.1.14 PlanItemCreationSupporter (Class)

This interface must be implemented in any modules that wish to support the plan module. The modules must attach their PlanItemCreationSupporters at startup. The GUIPlanModule will then call the supporter when it is time to display the Plan menu or to create a specific type of plan item or GUIPlanItem.

3.10.1.1.15 Response DataCreator (Class)

This interface enables the creation of type-specific ResponsePlanItemData objects, which are used for creating the appropriate type of ResponsePlanItem. An object implementing this interface can be added to the response plan of a traffic event. Implementers of this interface include plan items and response devices.

3.10.1.1.16 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.10.2 Sequence Diagrams

3.10.2.1 GUIPlanModule:AddPlan (Sequence Diagram)

This diagram shows how a new plan is added to the system. The user clicks on the Add Plan menu item in the GUIPlanNavGroup's context menu. (This menu item will only be displayed if the user has rights.) The GUIPlanNavGroup will create an uninitialized GUIPlan object with default properties and will call its doProperties method. This is a temporary object, used only for displaying the properties. The temporary GUIPlan will create a modeless PlanPropertiesDialog and display it. When the user clicks OK, the dialog will ask the GIUPlan to create a Plan from the properties entered from the dialog. A CommandStatus object is created to keep the user informed about the progress of the command. The GUIPlan queries the trader for all of the Plan Factories. It then tries to create the plan by calling each plan factory in the trader passing it the access token, until a factory successfully creates the Plan object. If the Plan was created, the PlanAdded event will be pushed from the plan server through the plan event channel to update all of the GUIs. See the GUIPlanModule:PlanAddedEvent diagram for more details.

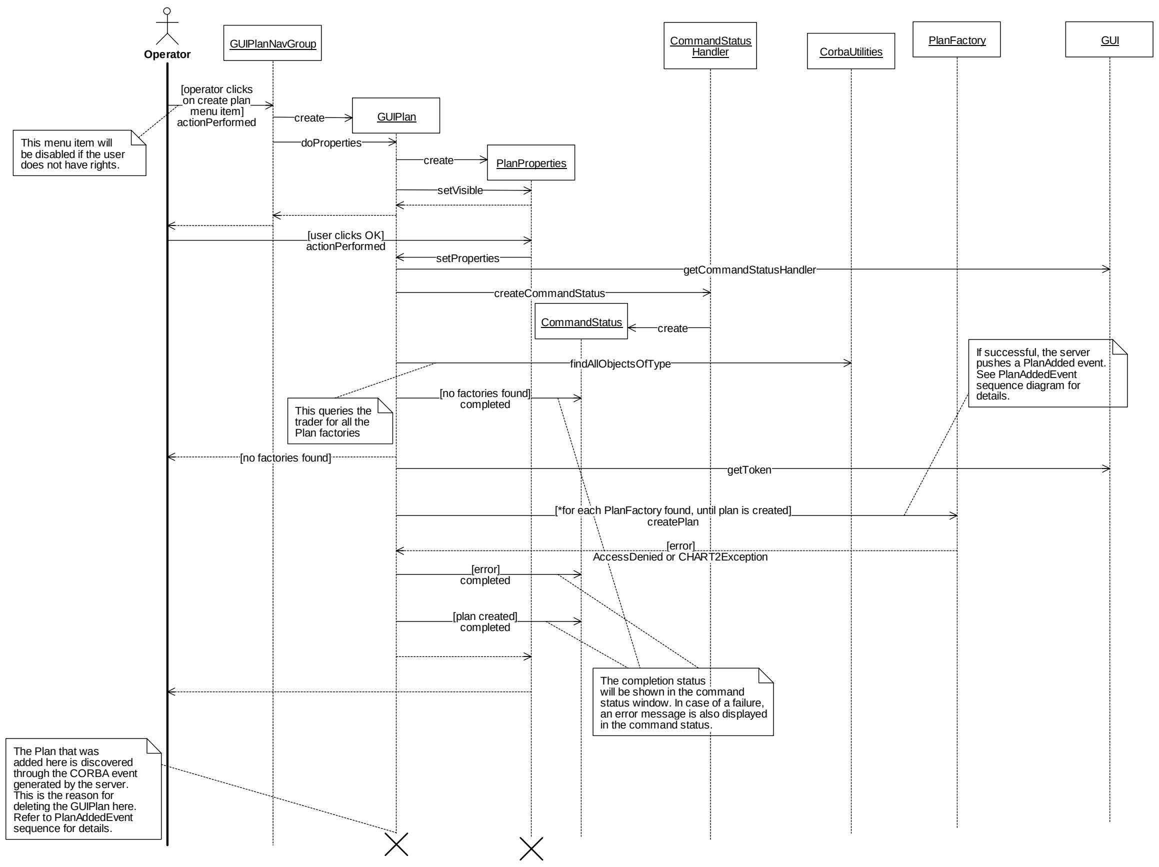
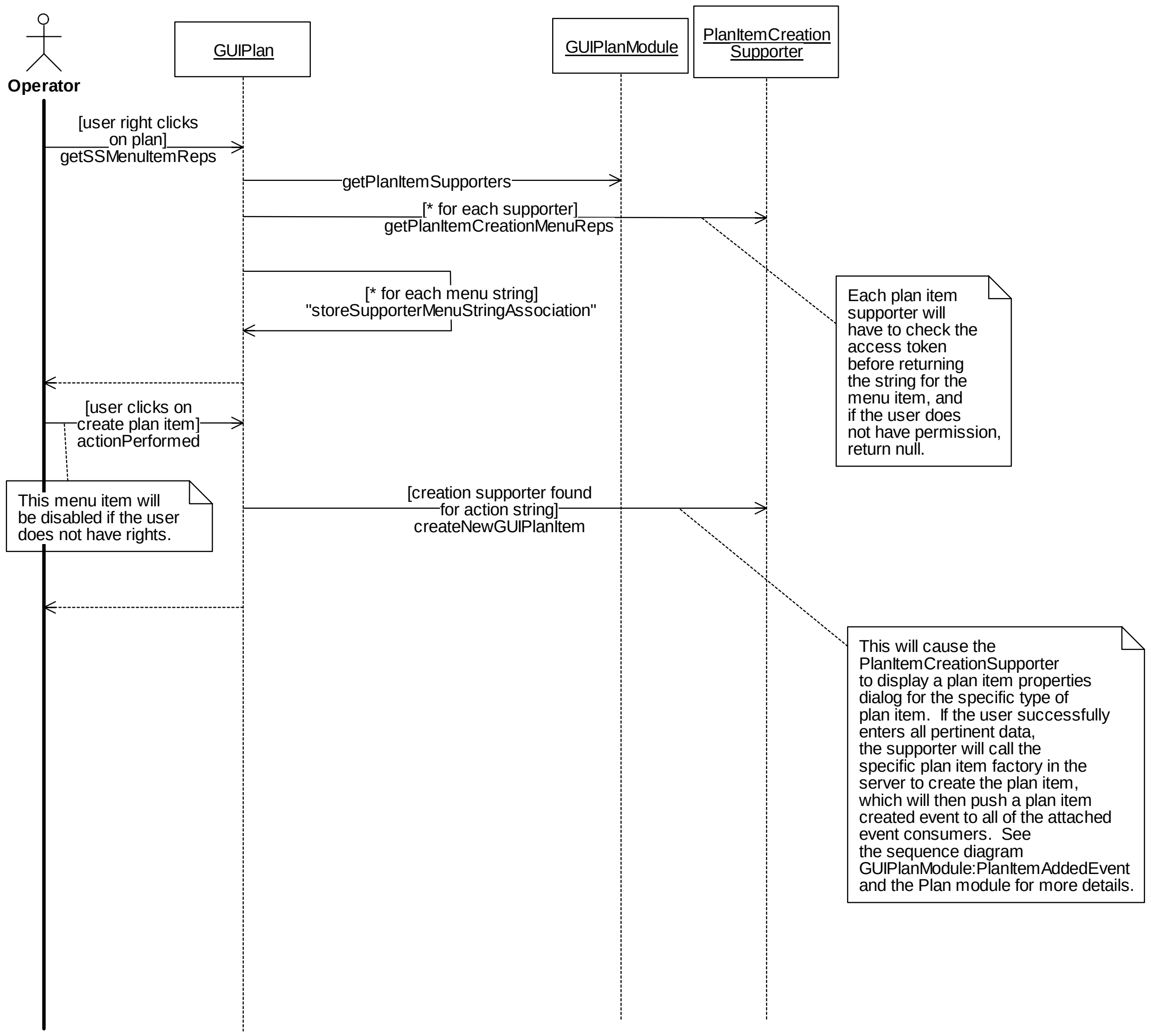

Operator
GUIPlanNavGroup
CommandStatus Handler
CorbaUtilities
PlanFactory
GUI
[operator clicks on create plan menu item] actionPerformed
create
GUIPlan
This menu item will be disabled if the user does not have rights.
doProperties
create
PlanProperties
setVisible
[user clicks OK] actionPerformed
setProperties
getCommandStatusHandler
createCommandStatus
CommandStatus
create
findAllObjectsOfType
If successful, the server pushes a PlanAdded event. See PlanAddedEvent sequence diagram for details.
[no factories found] completed
This queries the trader for all the Plan factories
[no factories found]
getToken
[*for each PlanFactory found, until plan is created] createPlan
[error] AccessDenied or CHART2Exception
[error] completed
[plan created] completed
The completion status will be shown in the command status window. In case of a failure, an error message is also displayed in the command status.
The Plan that was added here is discovered through the CORBA event generated by the server. This is the reason for deleting the GUIPlan here. Refer to PlanAddedEvent sequence for details.

Figure 117. GUIPlanModule:AddPlan (Sequence Diagram)

3.10.2.2 GUIPlanModule:CreatePlanItem (Sequence Diagram)

This diagram shows how a plan item is created. When the user invokes the menu on the GUIPlan object, the GUIPlan object asks the GUIPlanModule for all of the attached PlanItemCreationSupporters. It then asks each of the supporters for the strings to use for the plan item creation menu items. Each string is associated with the supporter that supplied it, and the associations are stored in the GUIPlan object for use when a menu item is clicked on. When the user clicks on one of these menu items, the GUIPlan's actionPerformed method will be called, and the GUIPlan will find the matching string stored in the association, and will call the corresponding PlanItemCreationSupporter to create the new plan item. See the modules that support plan item creation for more details on how plan items are created.



Operator
GUIPlan
GUIPlanModule
PlanItemCreation Supporter
[user right clicks on plan] getSSMenuItemReps
getPlanItemSupporters
[* for each supporter] getPlanItemCreationMenuReps
[* for each menu string] "storeSupporterMenuStringAssociation"
[user clicks on create plan item] actionPerformed
[creation supporter found for action string] createNewGUIPlanItem
Each plan item supporter will have to check the access token before returning the string for the menu item, and if the user does not have permission, return null.
This menu item will be disabled if the user does not have rights.
This will cause the PlanItemCreationSupporter to display a plan item properties dialog for the specific type of plan item. If the user successfully enters all pertinent data, the supporter will call the specific plan item factory in the server to create the plan item, which will then push a plan item created event to all of the attached event consumers. See the sequence diagram GUIPlanModule:PlanItemAddedEvent and the Plan module for more details.

Figure 118. GUIPlanModule:CreatePlanItem (Sequence Diagram)

3.10.2.3 GUIPlanModule:Discovery (Sequence Diagram)

This diagram shows what happens during the discovery process, in which the module has a chance to find out about event channels and objects. The GUI will periodically call the module, first to discover event channels and then to discover objects. During the event channel discovery phase, the module looks for Plan event channels in the trader. If it finds any, it creates a PushEventConsumer and attaches itself to the Event Consumer Group. This will attach the module to the event channel and will reattach it automatically if the event service is restarted. If the module was previously attached to the event channel, it will be ignored. During the object discovery phase, the GUI calls the module to discover objects. The module will query the Plan objects in the trader. If the Plan does not already exist in the DataModel, a new GUIPlan wrapper object will be created and added to the DataModel. When the GUIPlan object is created, it asks the Plan for all of its PlanItems. For each item that is not already in the DataModel, it will call all of the attached PlanItemCreationSupporters and ask each one to attempt to create the specific type of GUIPlanItem wrapper object for the generic PlanItem object. Each PlanItemCreationSupporter will check whether the generic PlanItem object is of its own specific class of plan item. If so, the PlanItemCreationSupporter must create an object of its own specific class of GUIPlanItem object to wrap the PlanItem. If a wrapper object was created, it will be added to the DataModel. After a short delay, the changes made through the DataModel will update any windows that are attached to the DataModel.

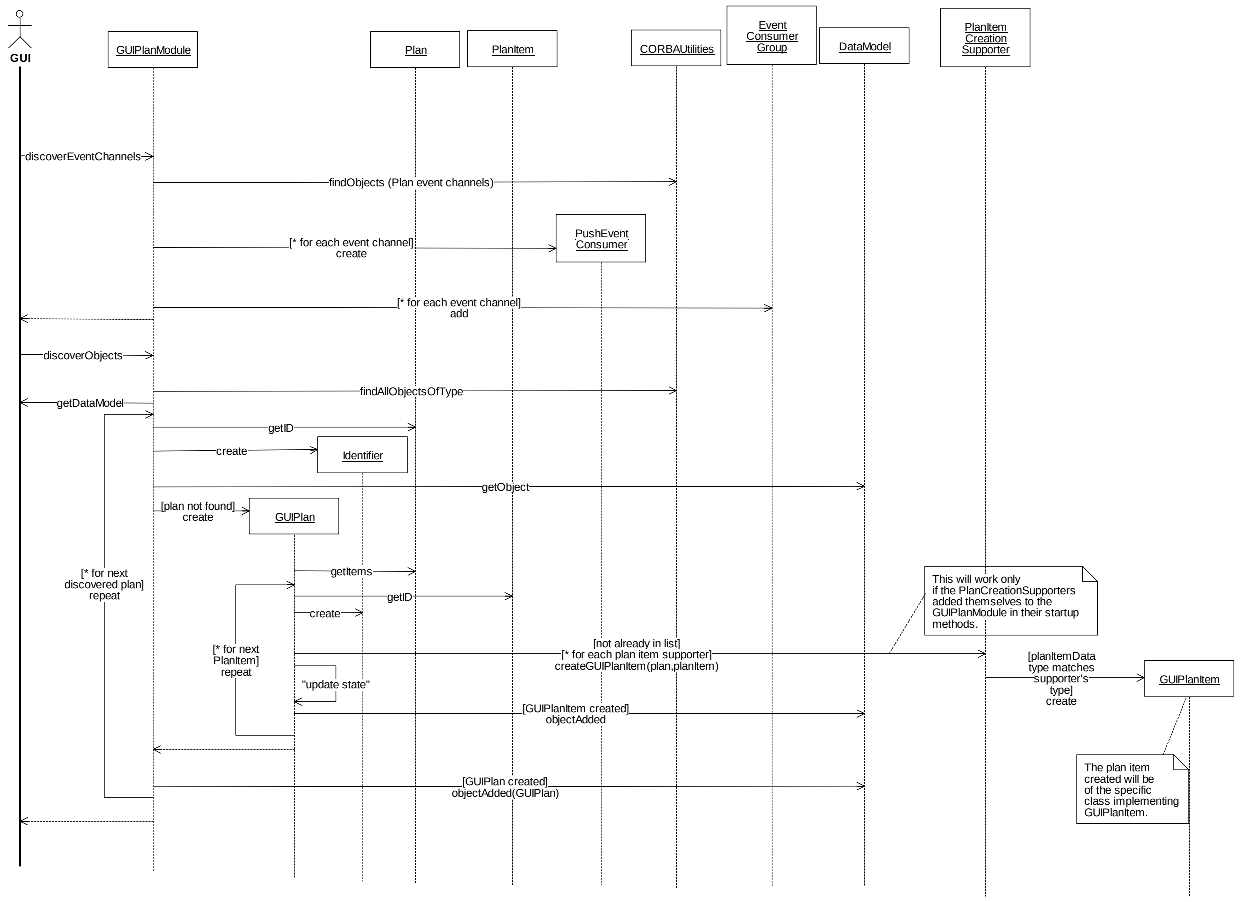

GUI
GUIPlanModule
Plan
PlanItem
CORBAUtilities
Event Consumer Group
DataModel
PlanItem Creation Supporter
discoverEventChannels
findObjects (Plan event channels)
[* for each event channel] create
PushEvent Consumer
[* for each event channel] add
discoverObjects
findAllObjectsOfType
getDataModel
getID
create
Identifier
getObject
[plan not found] create
GUIPlan
[* for next discovered plan] repeat
getItems
getID
create
[* for next PlanItem] repeat
[not already in list] [* for each plan item supporter] createGUIPlanItem(plan,planItem)
"update state"
[GUIPlanItem created] objectAdded
[planItemData type matches supporter's type] create
GUIPlanItem
This will work only if the PlanCreationSupporters added themselves to the GUIPlanModule in their startup methods.
[GUIPlan created] objectAdded(GUIPlan)
The plan item created will be of the specific class implementing GUIPlanItem.

Figure 119. GUIPlanModule:Discovery (Sequence Diagram)

3.10.2.4 GUIPlanModule:PlanAddedEvent (Sequence Diagram)

This diagram shows how the event is handled when a Plan is added. The GUIPlanModule makes sure that the GUIPlan does not already exist in the DataModel, and assuming it does not, it creates the GUIPlan wrapper object for the Plan. The GUIPlan object is then added to the DataModel and the GUIPlanNavGroup, and the DataModel will update all attached observers to show the change.

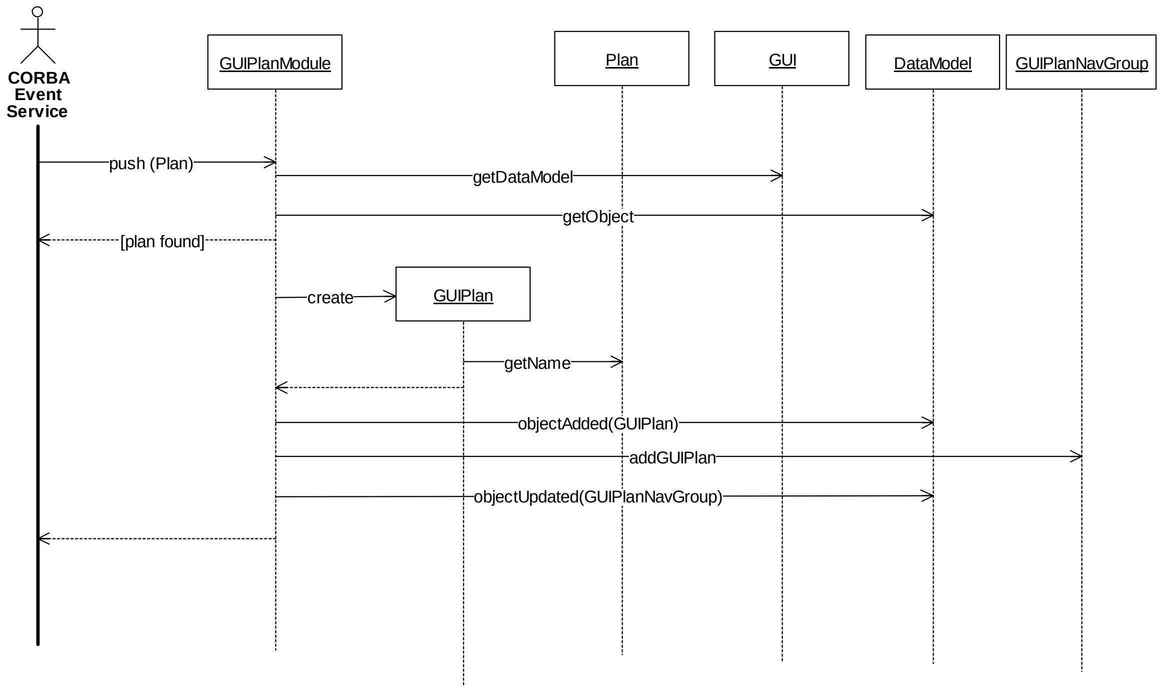


Figure 120. GUIPlanModule:PlanAddedEvent (Sequence Diagram)

3.10.2.5 GUIPlanModule:PlanItemAddedEvent (Sequence Diagram)

This diagram shows the handling of the event after a new PlanItem has been created. First, the GUIPlan to which the new PlanItem belongs is retrieved from the DataModel. Then the module will ask each PlanItemCreationSupporter to attempt to create a specific GUIPlanItem wrapper object if the generic PlanItem is a correct type for the supporter. If a GUIPlanItem object was created by one of the creation supporters, it is added to the GUIPlan and to the DataModel. The GUIPlan is also updated through the DataModel to make sure that any windows will be updated.

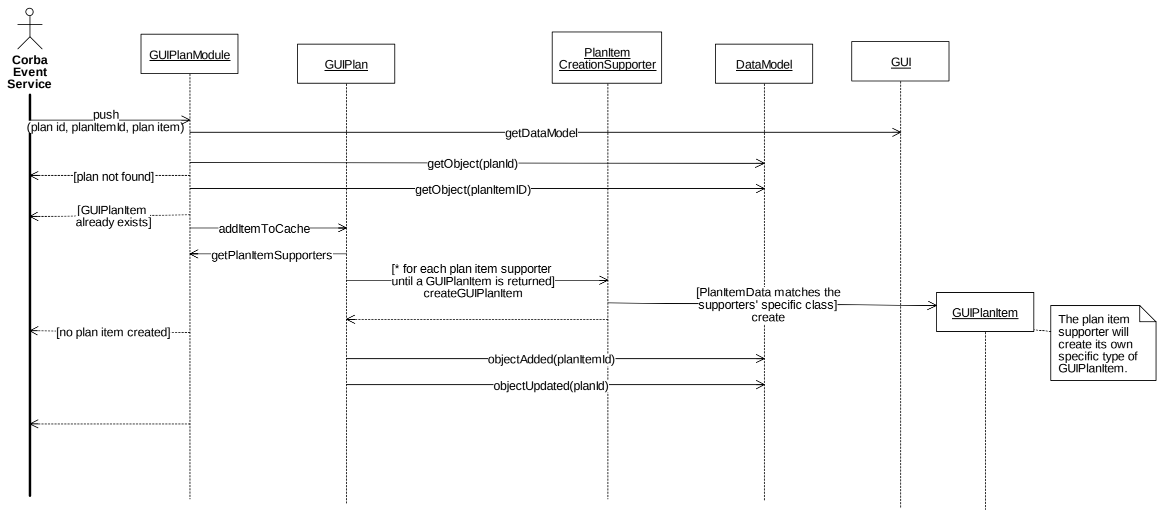

Corba Event Service
GUIPlanModule
GUIPlan
PlanItem CreationSupporter
DataModel
GUI
push (plan id, planItemId, plan item)
getDataModel
getObject(planId)
[plan not found]
getObject(planItemID)
[GUIPlanItem already exists]
addItemToCache
getPlanItemSupporters
[* for each plan item supporter until a GUIPlanItem is returned] createGUIPlanItem
[PlanItemData matches the supporters' specific class] create
GUIPlanItem
The plan item supporter will create its own specific type of GUIPlanItem.
[no plan item created]
objectAdded(planItemId)
objectUpdated(planId)

Figure 121. GUIPlanModule:PlanItemAddedEvent (Sequence Diagram)

3.10.2.6 GUIPlanModule:PlanItemRemovedEvent (Sequence Diagram)

This diagram shows how a PlanItemRemoved event is handled, after a plan item is deleted. The GUIPlanModule received the PlanItem identifier and looks up the GUIPlanItem object in the DataModel. If found, the module gets the GUIPlan and asks it to remove the GUIPlanItem from its collection. The GUIPlan object is then updated through the DataModel, and the GUIPlanItem is removed from the DataModel. Any attached observers (e.g., windows) will be updated after a short delay. The GUIPlanItem will then be removed from memory by Java when the observers remove their references to it.

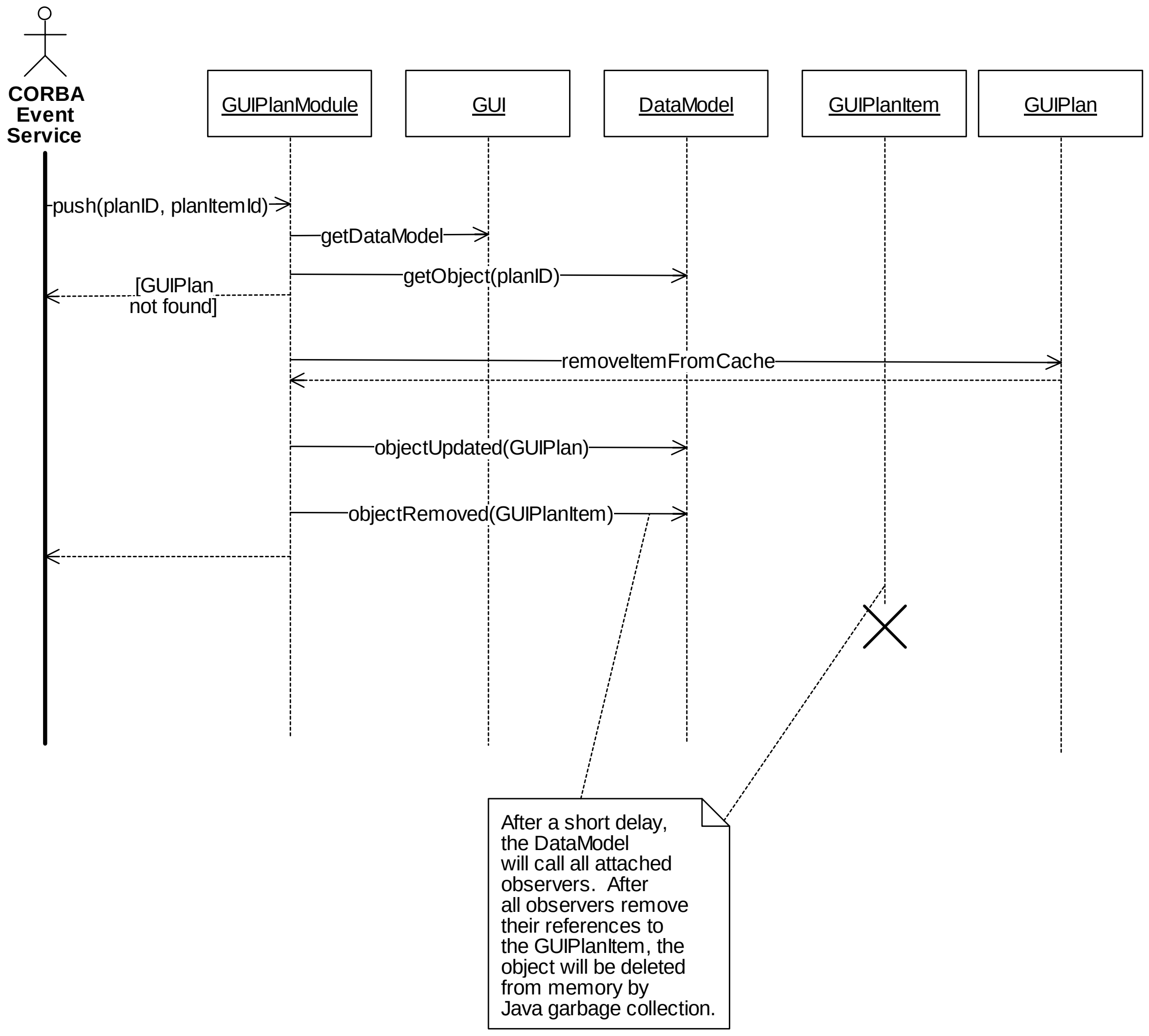

CORBA Event Service
GUIPlanModule
GUI
DataModel
GUIPlanItem
GUIPlan
push(planID, planItemId)
getDataModel
getObject(planID)
[GUIPlan not found]
removeItemFromCache
objectUpdated(GUIPlan)
objectRemoved(GUIPlanItem)
After a short delay, the DataModel will call all attached observers. After all observers remove their references to the GUIPlanItem, the object will be deleted from memory by Java garbage collection.

Figure 122. GUIPlanModule:PlanItemRemovedEvent (Sequence Diagram)

3.10.2.7 GUIPlanModule:PlanRemovedEvent (Sequence Diagram)

This diagram shows how a PlanRemoved event is handled. First, an attempt is made to get the GUIPlan object from the DataModel. If it exists, the GUIPlan is removed from the GUIPlanNavGroup. The GUIPlanNavGroup update notification is invoked through the DataModel, and the GUIPlan is removed from the DataModel. The DataModel will cause any attached observers to display the change.

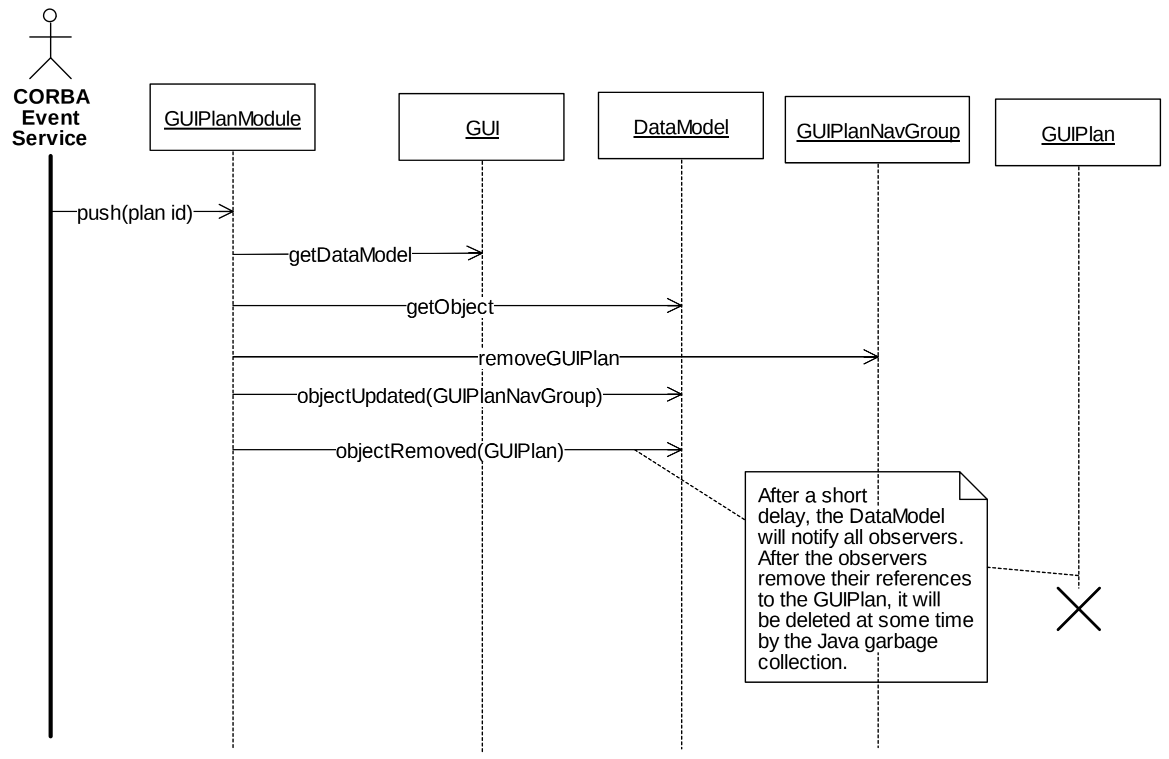

CORBA Event Service
GUIPlanModule
GUI
DataModel
GUIPlanNavGroup
GUIPlan
push(plan id)
getDataModel
getObject
removeGUIPlan
objectUpdated(GUIPlanNavGroup)
objectRemoved(GUIPlan)
After a short delay, the DataModel will notify all observers. After the observers remove their references to the GUIPlan, it will be deleted at some time by the Java garbage collection.

Figure 123. GUIPlanModule:PlanRemovedEvent (Sequence Diagram)

3.10.2.8 GUIPlanModule:RemovePlan (Sequence Diagram)

This diagram shows how a plan is removed from the system. The operator clicks on the Delete Plan menu item. A CommandStatus object is created to keep the user informed about the progress of the command. The GUIPlan then gets the access token and calls the Plan to remove itself. If successful, it will cause the server to push a PlanRemoved event to be pushed through the event channel. See the diagram GUIPlanModule:PlanRemovedEvent for details on how the GUIs are updated after the plan is removed.

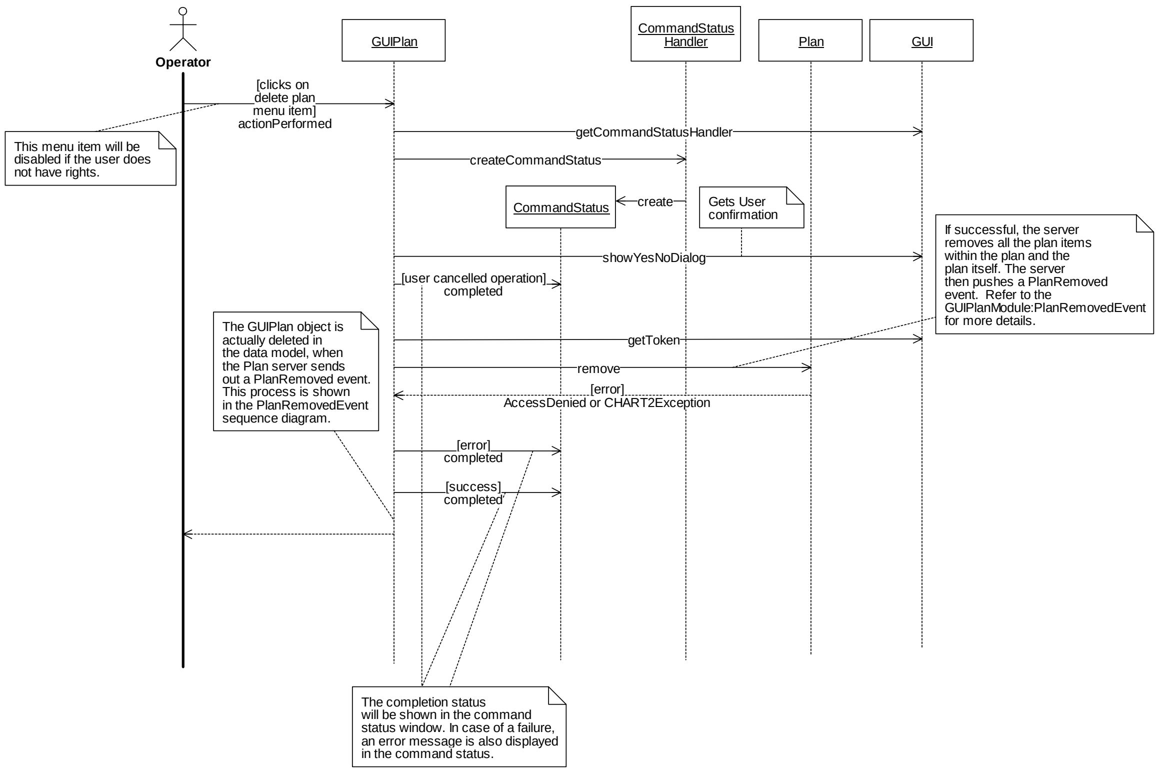
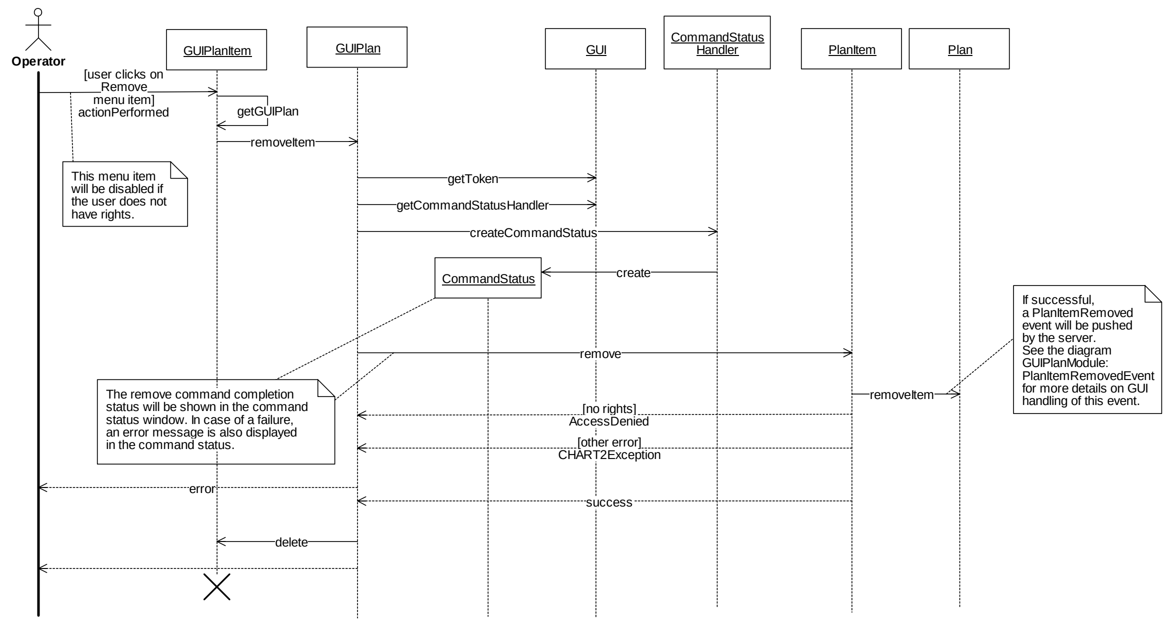

Operator
GUIPlan
CommandStatus Handler
Plan
GUI
[clicks on delete plan menu item] actionPerformed
This menu item will be disabled if the user does not have rights.
getCommandStatusHandler
createCommandStatus
CommandStatus
create
Gets User confirmation
If successful, the server removes all the plan items within the plan and the plan itself. The server then pushes a PlanRemoved event. Refer to the GUIPlanModule:PlanRemovedEvent for more details.
showYesNoDialog
[user cancelled operation] completed
The GUIPlan object is actually deleted in the data model, when the Plan server sends out a PlanRemoved event. This process is shown in the PlanRemovedEvent sequence diagram.
getToken
remove
[error] AccessDenied or CHART2Exception
[error] completed
[success] completed
The completion status will be shown in the command status window. In case of a failure, an error message is also displayed in the command status.

Figure 124. GUIPlanModule:RemovePlan (Sequence Diagram)

3.10.2.9 GUIPlanModule:RemovePlanItem (Sequence Diagram)

This diagram shows how a plan item is removed from a plan and deleted. The operator selects the plan item and invokes the item's context menu, then clicks on Delete Item. The GUIPlanItem calls the GUIPlan that it is contained in to remove the item. The GUIPlan then calls the Plan to remove the item. The served Plan object will then remove the item and push a PlanItemRemoved event through the event channel. See the diagram GUIPlanModule:PlanItemRemovedEvent for more details on this event.



Operator
GUIPlanItem
GUIPlan
GUI
CommandStatus Handler
PlanItem
Plan
[user clicks on Remove menu item] actionPerformed
getGUIPlan
removeItem
This menu item will be disabled if the user does not have rights.
getToken
getCommandStatusHandler
createCommandStatus
CommandStatus
create
If successful, a PlanItemRemoved event will be pushed by the server. See the diagram GUIPlanModule: PlanItemRemovedEvent for more details on GUI handling of this event.
remove
removeItem
The remove command completion status will be shown in the command status window. In case of a failure, an error message is also displayed in the command status.
[no rights] AccessDenied
[other error] CHART2Exception
error
success
delete

Figure 125. GUIPlanModule:RemovePlanItem (Sequence Diagram)

3.10.2.10 GUIPlanModule:Shutdown (Sequence Diagram)

When the GUI calls the module's shutdown method, the module deactivates from the POA to clean up.

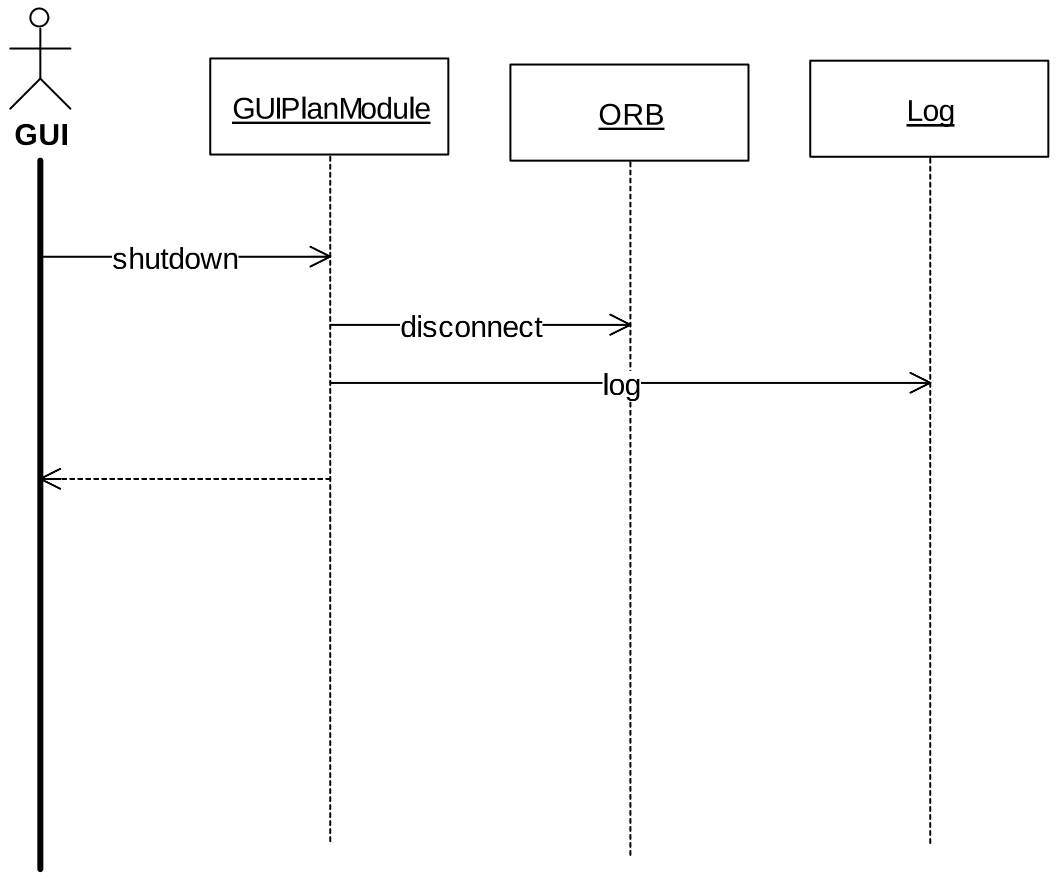


Figure 126. GUIPlanModule:Shutdown (Sequence Diagram)

3.10.2.11 GUIPlanModule:Startup (Sequence Diagram)

The startup for the GUIPlanModule begins when the GUI calls the startup method. At this time the module activates itself with the POA so that it can be called as a PushConsumer. It also creates a Navigator group to hold the GUIPlan objects and adds the group to the DataModel. NOTE: Any modules deemed necessary to support plan item creation should attach themselves to the GUIPlanModule in their startup methods.

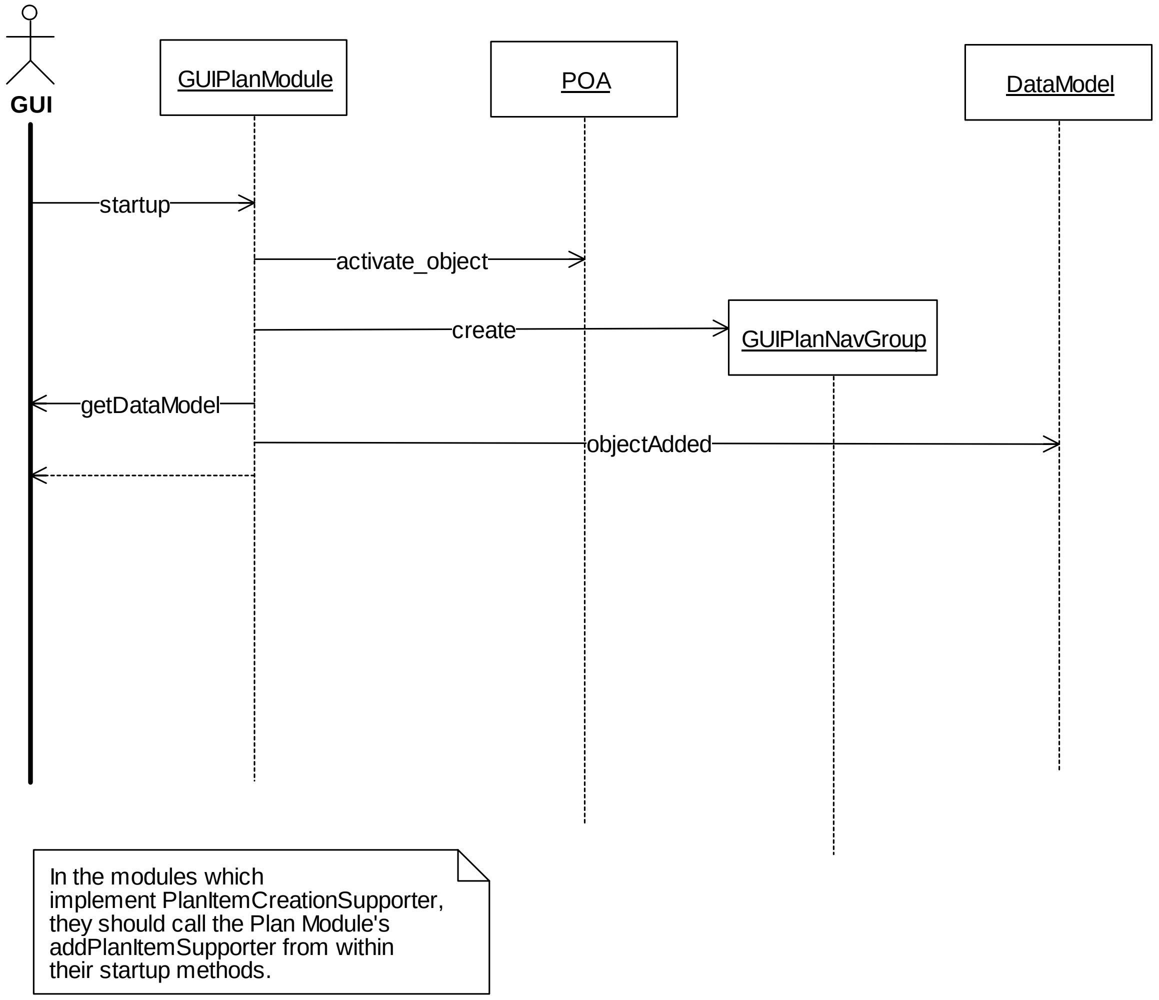

GUI
GUIPlanModule
POA
DataModel
startup
activate_object
create
GUIPlanNavGroup
getDataModel
objectAdded
In the modules which implement PlanItemCreationSupporter, they should call the Plan Module's addPlanItemSupporter from within their startup methods.

Figure 127. GUIPlanModule:Startup (Sequence Diagram)

3.11 GUIResourcesModule

3.11.1 Class Diagrams

3.11.1.1 GUIResourcesModuleClasses (Class Diagram)

This diagram represents the classes used by the GUI Resources module and their relationships.

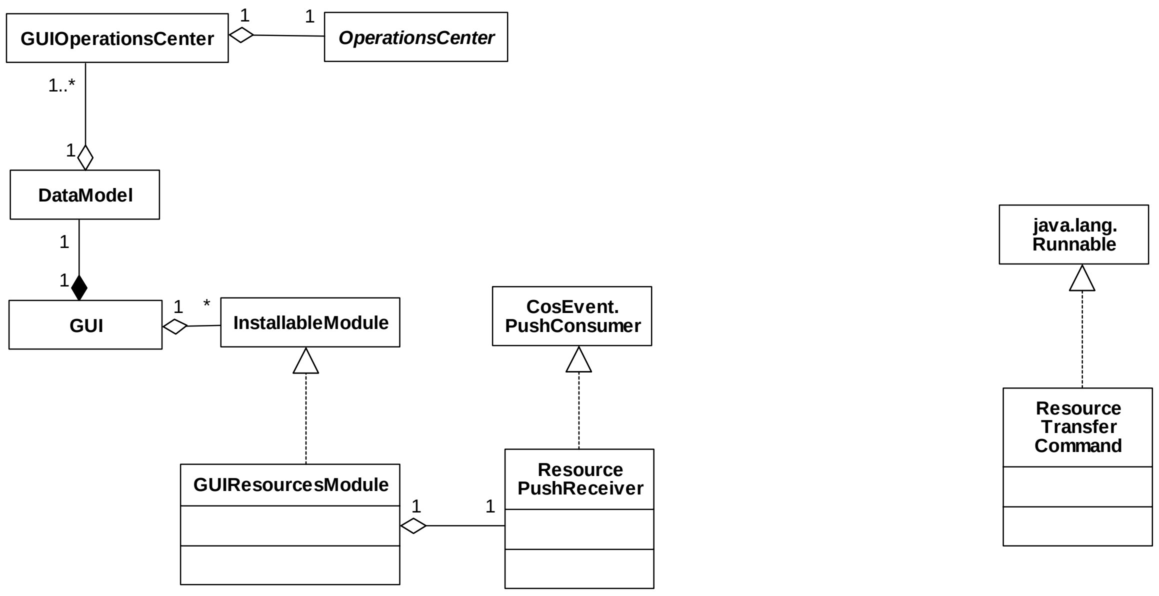


Figure 128. GUIResourcesModuleClasses (Class Diagram)

3.11.1.1.1 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.11.1.1.2 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.11.1.1.3 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.11.1.1.4 GUIResourcesModule (Class)

This class is an installable GUI module that handles all of the resource-specific functionality in the GUI.

3.11.1.1.5 java.lang. Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.11.1.1.6 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.11.1.1.7 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

3.11.1.1.8 GUIOperationsCenter (Class)

This class is a GUI "wrapper" object that is used to wrap a OperationsCenter object. The wrapping is done to cache the data locally for faster access, and to provide GUI-specific functionalities to the wrapped object.

3.11.1.1.9 Resource PushReceiver (Class)

This class receives CORBA events from the Resource event channel and deals with each type of event as appropriate.

3.11.1.1.10 Resource Transfer Command (Class)

This command object is invoked on the main AWT event thread after an UnhandledControlledResources CORBA event is received. This enables the TransferResources dialog to be invoked from the event thread.

3.11.2 Sequence Diagrams

3.11.2.1 GUIResourcesModule:Discovery (Sequence Diagram)

This diagram shows how the Resources event channels and Operations Center objects are discovered and added to the system. This will be a periodic process, and the GUI will call the GUIResourcesModule repeatedly. When the GUI asks the module to discover event channels, it looks up the Resource event channels in the trader. It then creates a PushEventConsumer and adds it to the EventConsumerGroup, which actually attaches the consumer to the channel and reattaches it if the event service is restarted. (Duplicate channels will be ignored). The GUI then calls the module to discover objects.

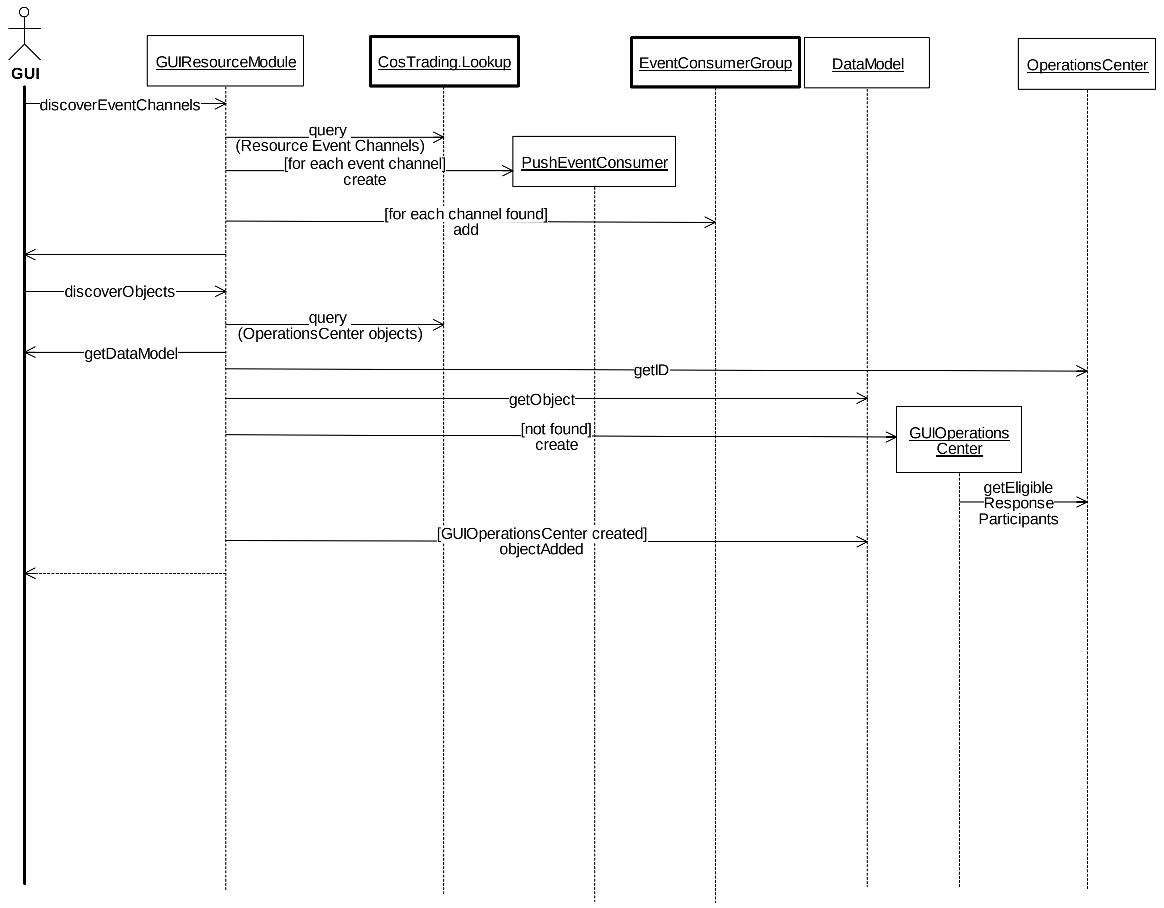


Figure 129. GUIResourcesModule:Discovery (Sequence Diagram)

3.11.2.2 GUIResourcesModule:EventHandling (Sequence Diagram)

The EventHandling diagram shows how the GUI receives update information from the CORBA Event service and displays it to the user once the DataModel is updated.

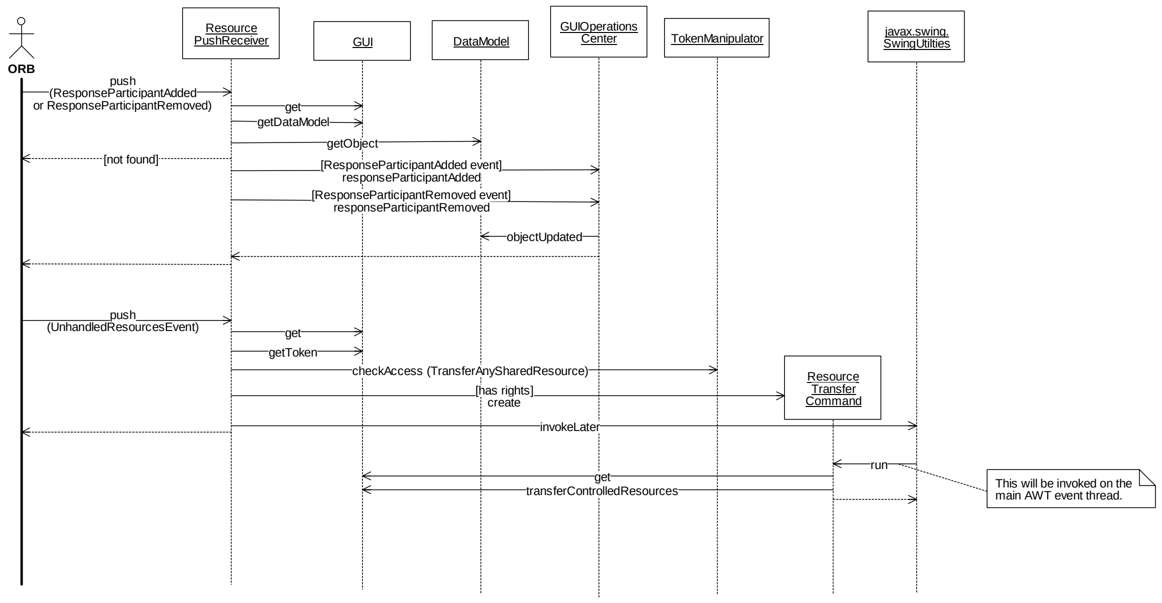

ORB
Resource PushReceiver
GUI
DataModel
GUIOperations Center
TokenManipulator
javax.swing. SwingUtilities
push (ResponseParticipantAdded or ResponseParticipantRemoved)
get
getDataModel
getObject
[not found]
[ResponseParticipantAdded event] responseParticipantAdded
[ResponseParticipantRemoved event] responseParticipantRemoved
objectUpdated
push (UnhandledResourcesEvent)
get
getToken
checkAccess (TransferAnySharedResource)
Resource Transfer Command
[has rights] create
invokeLater
run
get
transferControlledResources
This will be invoked on the main AWT event thread.

Figure 130. GUIResourcesModule:EventHandling (Sequence Diagram)

3.11.2.3 GUIResourcesModule:Login (Sequence Diagram)

This diagram shows what happens in the GUIResourcesModule when the user logs in to the system.

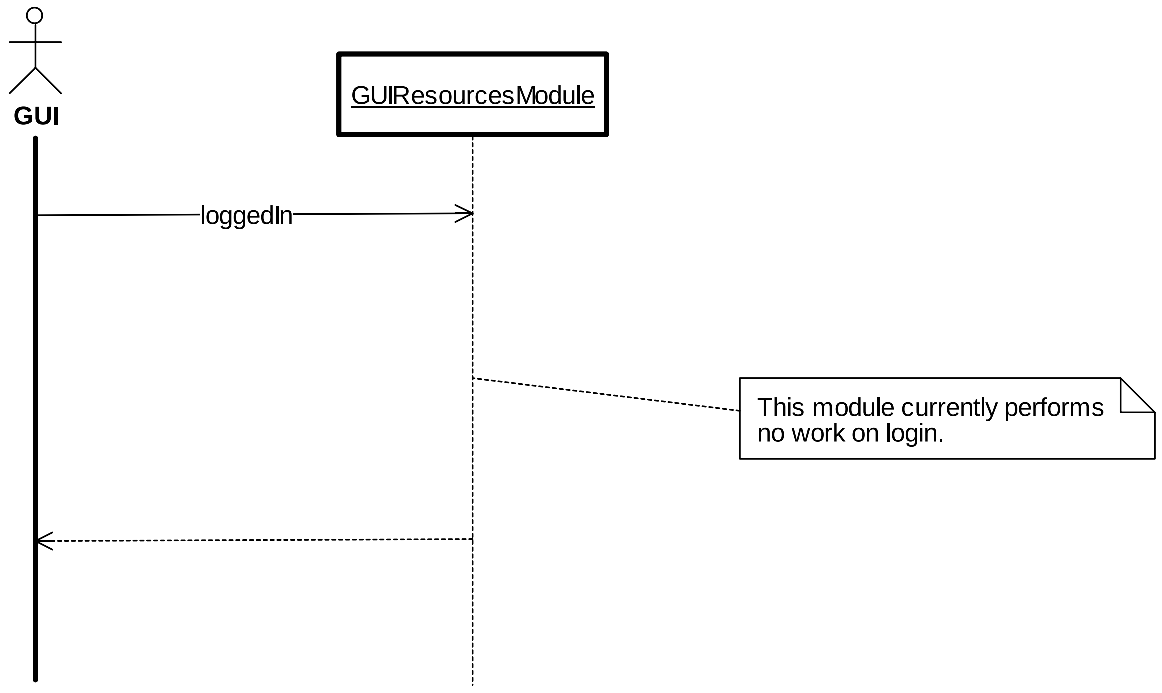


Figure 131. GUIResourcesModule:Login (Sequence Diagram)

3.11.2.4 GUIResourcesModule:Logout (Sequence Diagram)

This diagram shows what happens when the user logs out. The GUI calls all of the InstallableModule objects' loggedOut() methods, but the GUIResourcesModule currently does nothing during logout.

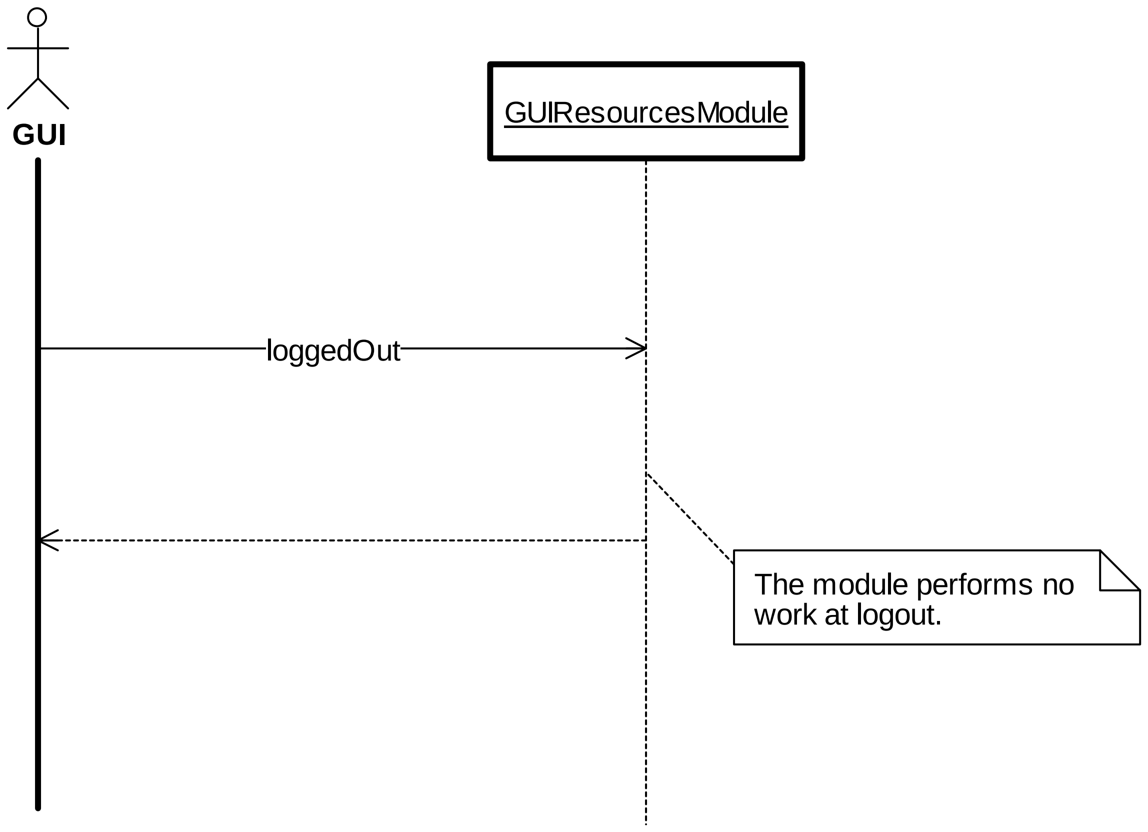


Figure 132. GUIResourcesModule:Logout (Sequence Diagram)

3.11.2.5 GUIResourcesModule:Shutdown (Sequence Diagram)

This diagram shows what happens when the GUI shuts down. The GUI calls all of the InstallableModule objects' shutdown() methods, and the GUIResourcesModule uses this method to disconnect itself from the ORB.

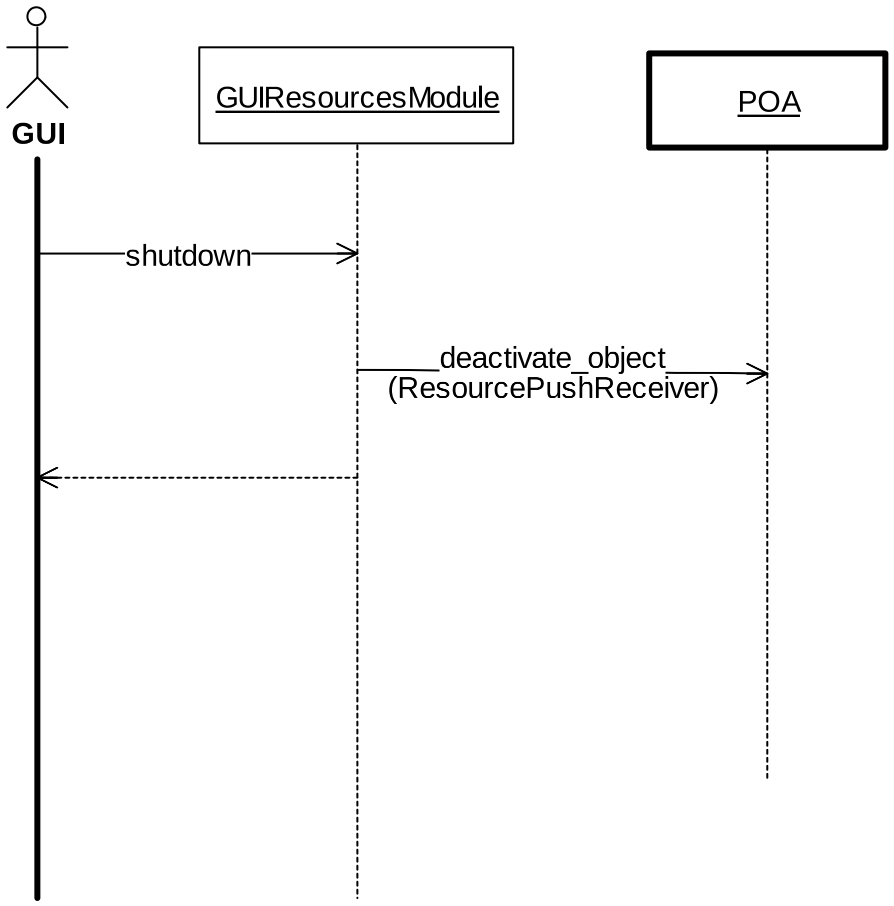


Figure 133. GUIResourcesModule:Shutdown (Sequence Diagram)

3.11.2.6 GUIResourcesModule:Startup (Sequence Diagram)

This diagram shows the steps taken to initialize the GUIResourcesModule. At startup time, the GUI calls the startup method in each of the installable modules. When the GUIResourcesModule startup is called, the ORB connections are made in order to receive the Resource events from the server.

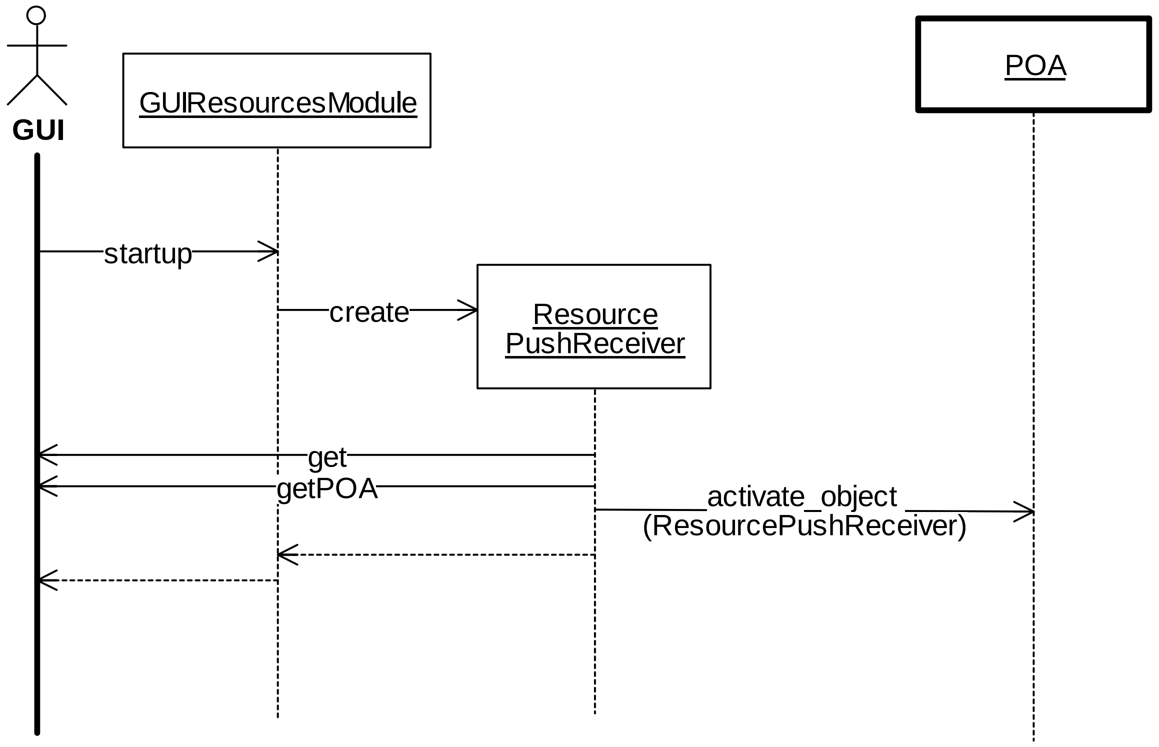


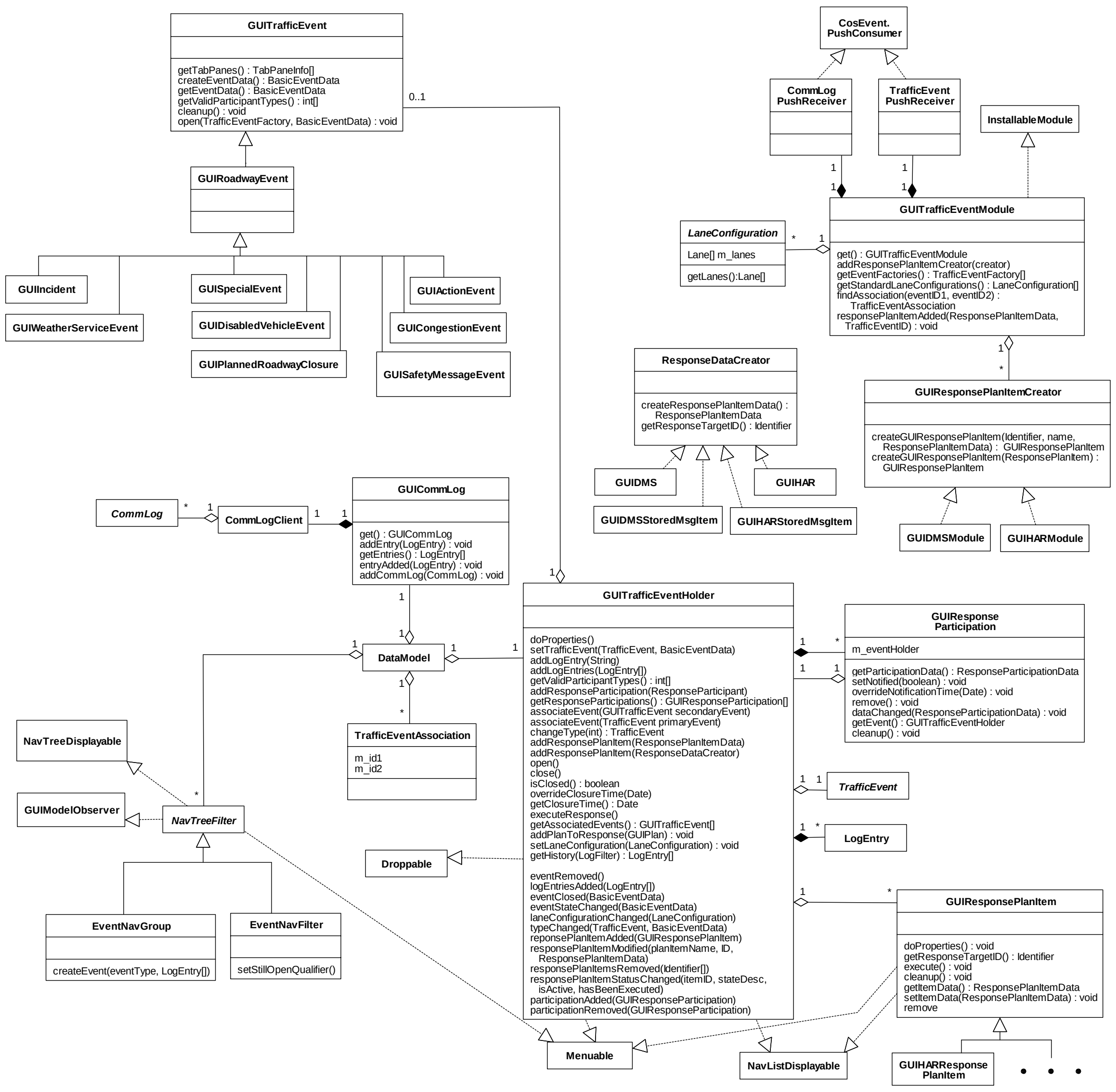
Figure 134. GUIResourcesModule:Startup (Sequence Diagram)

3.12 GUITrafficEventModule

3.12.1 Class Diagrams

3.12.1.1 GUITrafficEventModuleClasses (Class Diagram)

This diagram shows the overall architecture of the GUITrafficEventModule. The utility classes and dialogs are shown on other diagrams. The GUIResponseParticipation class hierarchy is in the GUITrafficEventModuleUtilityClasses diagram due to a lack of space on this diagram.



GUITrafficEvent
getTabPanes() : TabPaneInfo[]
createEventData() : BasicEventData
getEventData() : BasicEventData
getValidParticipantTypes() : int[]
cleanup() : void
open(TrafficEventFactory, BasicEventData) : void
0..1
GUIRoadwayEvent
GUIIncident
GUIWeatherServiceEvent
GUISpecialEvent
GUIDisabledVehicleEvent
GUIPlannedRoadwayClosure
GUIActionEvent
GUICongestionEvent
GUISafetyMessageEvent
CosEvent.
PushConsumer
CommLog
PushReceiver
TrafficEvent
PushReceiver
InstallableModule
1
1
1
1
GUITrafficEventModule
get() : GUITrafficEventModule
addResponsePlanItemCreator(creator)
getEventFactories() : TrafficEventFactory[]
getStandardLaneConfigurations() : LaneConfiguration[]
findAssociation(eventID1, eventID2) :
TrafficEventAssociation
responsePlanItemAdded(ResponsePlanItemData,
TrafficEventID) : void
LaneConfiguration
Lane[] m_lanes
getLanes():Lane[]
*
1
1
*
ResponseDataCreator
createResponsePlanItemData() :
ResponsePlanItemData
getResponseTargetID() : Identifier
GUIDMS
GUIHAR
GUIDMSStoredMsgItem
GUIHARStoredMsgItem
GUIResponsePlanItemCreator
createGUIResponsePlanItem(Identifier, name,
ResponsePlanItemData) : GUIResponsePlanItem
createGUIResponsePlanItem(ResponsePlanItem) :
GUIResponsePlanItem
GUIDMSModule
GUIHARModule
CommLog
*
1
CommLogClient
1
1
GUICommLog
get() : GUICommLog
addEntry(LogEntry) : void
getEntries() : LogEntry[]
entryAdded(LogEntry) : void
addCommLog(CommLog) : void
1
1
1
1
1
DataModel
1
*
TrafficEventAssociation
m_id1
m_id2
NavTreeDisplayable
GUIModelObserver
*
NavTreeFilter
EventNavGroup
createEvent(eventType, LogEntry[])
EventNavFilter
setStillOpenQualifier()
Droppable
1
GUITrafficEventHolder
doProperties()
setTrafficEvent(TrafficEvent, BasicEventData)
addLogEntry(String)
addLogEntries(LogEntry[])
getValidParticipantTypes() : int[]
addResponseParticipation(ResponseParticipant)
getResponseParticipations() : GUIResponseParticipation[]
associateEvent(GUITrafficEvent secondaryEvent)
associateEvent(TrafficEvent primaryEvent)
changeType(int) : TrafficEvent
addResponsePlanItem(ResponsePlanItemData)
addResponsePlanItem(ResponseDataCreator)
open()
close()
isClosed() : boolean
overrideClosureTime(Date)
getClosureTime() : Date
executeResponse()
getAssociatedEvents() : GUITrafficEvent[]
addPlanToResponse(GUIPlan) : void
setLaneConfiguration(LaneConfiguration) : void
getHistory(LogFilter) : LogEntry[]
eventRemoved()
logEntriesAdded(LogEntry[])
eventClosed(BasicEventData)
eventStateChanged(BasicEventData)
laneConfigurationChanged(LaneConfiguration)
typeChanged(TrafficEvent, BasicEventData)
reponsePlanItemAdded(GUIResponsePlanItem)
responsePlanItemModified(planItemName, ID,
ResponsePlanItemData)
responsePlanItemsRemoved(Identifier[])
responsePlanItemStatusChanged(itemID, stateDesc,
isActive, hasBeenExecuted)
participationAdded(GUIResponseParticipation)
participationRemoved(GUIResponseParticipation)
1
*
1
1
GUIResponse
Participation
m_eventHolder
getParticipationData() : ResponseParticipationData
setNotified(boolean) : void
overrideNotificationTime(Date) : void
remove() : void
dataChanged(ResponseParticipationData) : void
getEvent() : GUITrafficEventHolder
cleanup() : void
1
1
TrafficEvent
1
*
LogEntry
1
*
GUIResponsePlanItem
doProperties() : void
getResponseTargetID() : Identifier
execute() : void
cleanup() : void
getItemData() : ResponsePlanItemData
setItemData(ResponsePlanItemData) : void
remove
Menuable
NavListDisplayable
GUIHARResponse
PlanItem
• • •

Figure 135. GUITrafficEventModuleClasses (Class Diagram)

3.12.1.1.1 CommLog (Class)

This class manages log entries. These can be general Communications Log entries or specific log entries for a specific Traffic Event. This class is the primary interface for the CommLog service. It is used to persist log entries in the CHART II system and retrieve them for review. Log entries can be created directly by users or indirectly as a result of manipulating Traffic Events.

3.12.1.1.2 CommLog PushReceiver (Class)

This class will receive and handle any CORBA events that are pushed by a server via the CORBA event service. This class will listen specifically for CORBA events sent through the Comm Log event channel.

3.12.1.1.3 CommLogClient (Class)

This class is a wrapper to be used by clients of the Communications Log. It provides services such as discovering instances of the CommLog in the trader and caching entries to the comm log that are added when the comm log is not available.

3.12.1.1.4 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.12.1.1.5 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

3.12.1.1.6 Droppable (Class)

This interface must be implemented by any object wishing to take part in a drag and drop operation. It is used by the DropHandler class to determine if a drop action should be allowed and to delegate the handling of the drop action after it is performed.

3.12.1.1.7 GUIResponse Participation (Class)

This class represents one instance of a participant (person, mobile unit, device, resource, or response team) being notified and involved in the response to an event. This is a wrapper object which wraps a ResponseParticipation CORBA interface object, caches the CORBA object's data, and/or adds GUI-specific functionality.

3.12.1.1.8 EventNavFilter (Class)

This Navigator filter allows the user to filter the Traffic Events shown in the Navigator based on Traffic Event-specific criteria. For example, it allows them to view events that were opened and/or closed at specified times. (Note: events that have been removed from the system are not available to be filtered and will not be displayed in the navigator regardless of the dates given in the filter).

3.12.1.1.9 EventNavGroup (Class)

This class is a singleton Navigator filter that shows all of the GUITrafficEventHolder objects in the system, and provides functionality so that the user can right click on the Navigator group to create a new traffic event in the system.

3.12.1.1.10 GUICongestionEvent (Class)

This is a wrapper class that wraps the CongestionEvent CORBA interface. It will cache the CongestionEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a CongestionEvent.

3.12.1.1.11 GUIDisabledVehicleEvent (Class)

This is a wrapper class that wraps the DisabledVehicleEvent CORBA interface. It will cache the DisabledVehicleEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a DisabledVehicleEvent.

3.12.1.1.12 GUIHARResponse PlanItem (Class)

This class provides a GUI "wrapper" object that is used to wrap a ResponsePlanItem CORBA interface that contains HAR-specific data and to supply GUI-specific functionality.

3.12.1.1.13 GUIIncident (Class)

This is a wrapper class that wraps the Incident CORBA interface. It will cache the Incident data and supply any type-specific GUI functionality related to the TrafficEvent being an Incident.

3.12.1.1.14 GUIResponsePlanItem (Class)

This is a base class for the GUI wrapper object that is used to wrap a ResponsePlanItem. The ResponsePlanItem represents a proposed action to perform on a target object in response to a TrafficEvent. This wrapper object adds GUI-specific functionality to the response plan item.

3.12.1.1.15 GUIResponsePlanItemCreator (Class)

This interface is used to enable the creation of specific types of GUIResponsePlanItem wrapper objects depending upon which type of ResponsePlanItem is being wrapped. Any class wishing to create GUIResponsePlanItems must implement this interface and add themselves to the GUITrafficEventModule at GUI startup time. When the GUITrafficEventModule discovers a ResponsePlanItem or catches a CORBA event indicating that a new response plan item has been created, it will call each known GUIResponsePlanItemCreator to give it an opportunity to create a specific type of GUI wrapper object.

3.12.1.1.16 GUIRoadwayEvent (Class)

This class extends the GUITrafficEvent class, adding any functionality that may be specific to the event being located on a roadway.

3.12.1.1.17 GUITrafficEventHolder (Class)

This object represents a TrafficEvent and provides GUI functionality for the TrafficEvent. This class contains generic data and operations that apply to any type of TrafficEvent. It also “holds” a type-specific GUITrafficEvent. If the type of the TrafficEvent is changed, the old GUITrafficEvent object (stored within this “holder” class) will be switched out for a new GUITrafficEvent of a different type, but the GUITrafficEventHolder will remain in existence.

3.12.1.1.18 GUITrafficEventModule (Class)

This class is an installable module within the GUI’s module framework. It provides the framework for all of the CHART Traffic Event and Comm Log functionality to be launched from the GUI. There can be at most one instance of a GUITrafficEventModule object within the GUI.

3.12.1.1.19 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.12.1.1.20 ResponseDataCreator (Class)

This interface enables the creation of type-specific ResponsePlanItemData objects, which are used for creating the appropriate type of ResponsePlanItem. An object implementing this interface can be added to the response plan of a traffic event. Implementers of this interface include plan items and response devices.

3.12.1.1.21 GUIActionEvent (Class)

This is a wrapper class that wraps the ActionEvent CORBA interface. It will cache the ActionEvent data and supply any type-specific GUI functionality related to the TrafficEvent being an ActionEvent.

3.12.1.1.22 GUIPlannedRoadwayClosure (Class)

This is a wrapper class that wraps the PlannedRoadwayClosure CORBA interface. It will cache the PlannedRoadwayClosure data and supply any type-specific GUI functionality related to the TrafficEvent being a PlannedRoadwayClosure.

3.12.1.1.23 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

3.12.1.1.24 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

3.12.1.1.25 GUITrafficEvent (Class)

This class is a base class for the wrappers that wrap the TrafficEvent CORBA interface. The implementing object will exist from when the TrafficEvent is created until the GUI is shut down, the TrafficEvent type is changed, or the TrafficEvent is removed from the system. The class may cache the TrafficEvent data, and italso provides GUI functionality for the specific type of TrafficEvent.

3.12.1.1.26 GUISafetyMessageEvent (Class)

This is a wrapper class that wraps the SafetyMessageEvent CORBA interface. It will cache the SafetyMessageEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a SafetyMessageEvent.

3.12.1.1.27 GUIWeatherServiceEvent (Class)

This is a wrapper class that wraps the WeatherServiceEvent CORBA interface. It will cache the WeatherServiceEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a WeatherServiceEvent.

3.12.1.1.28 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.12.1.1.29 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.12.1.1.30 GUISpecialEvent (Class)

This is a wrapper class that wraps the SpecialEvent CORBA interface. It will cache the SpecialEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a SpecialEvent.

3.12.1.1.31 TrafficEvent (Class)

Objects of this type represent traffic events that require action from system operators.

3.12.1.1.32 NavTreeFilter (Class)

This class serves as a node in the Navigator tree and filters objects to be displayed in the Navigator. It is an observer to the DataModel so that it can create the NavTreeFilteredObjectInstance objects for any NavTreeDisplayables that it contains. (Multiple instances can appear to represent one NavTreeDisplayable object). Filters can be cascaded to achieve a cumulative filtering effect; that is, a filter appearing under a parent filter will call the parent filter first to filter the objects, and then it will apply its own filtering method. The cascading of filters is therefore an “AND” operation. A filter can either be a system filter or a user-specific filter. System filters can only be modified by someone with the correct administrative rights, and they can only be added as a child of other system filters.

3.12.1.1.33 GUICommLog (Class)

This class is a wrapper for the CommLog CORBA interface object, and provides the GUI functionality for interacting with the Comm Log. No more than one instance of a GUICommLog object will exist within the GUI.

3.12.1.1.34 GUIDMSStoredMsgItem (Class)

This class is a GUI “wrapper” object that is used to wrap a PlanItem object which contains the DMSPlanItemData. It helps in the creation of a DMS plan item data using the DMSStoredMsgItemProperties object.

3.12.1.1.35 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

3.12.1.1.36 TrafficEventAssociation (Class)

This object is used to denote an association between two TrafficEvent objects. It stores only the IDs of the TrafficEvent objects because traffic event objects can be removed from the system, in which case the reference to the removed object would cause problems. The association object is separate to reduce complexity of maintaining bidirectional references between the events.

3.12.1.1.37 GUIDMS (Class)

This class is a GUI “wrapper” object that is used to wrap a CHART2DMS object. This is an abstract class that needs to be extended by the GUI DMS model specific classes.

3.12.1.1.38 GUIDMSModule (Class)

The GUIDMSModule is an installable module in the GUI that handles all of the DMS specific functionality. Only one GUIDMSModule object may exist within the GUI. This class implements the interfaces to support the frameworks of the GUIPlanModule, the GUILibraryModule, and the GUITrafficEventModule. It handles the creation of model specific GUI DMS objects using the model supporters.

3.12.1.1.39 LaneConfiguration (Class)

This class contains data that represents the configuration of the lanes.

3.12.1.1.40 TrafficEvent PushReceiver (Class)

This class will receive and handle any CORBA events that are pushed by a server via the CORBA event service. This class will listen specifically for CORBA events sent through the Traffic Event event channel.

3.12.1.1.41 GUIHAR (Class)

This class provides a GUI “wrapper” object that is used to wrap the CHART2HAR CORBA interface and to supply GUI-specific functionality.

3.12.1.1.42 GUIHARModule (Class)

The GUIHARModule is an installable module in the GUI, and provides all functionality specific to HAR and SHAZAM control. It requires that the GUIPlanModule, the GUILibraryModule, and the GUITrafficEventModule all be installed. Only one GUIHARModule object may exist within the GUI. This class implements the interfaces to support the frameworks of the GUIPlanModule, the GUILibraryModule, and the GUITrafficEventModule.

3.12.1.1.43 GUIHARStoredMsgItem (Class)

This class provides a GUI “wrapper” object that is used to wrap the HARStoredMsgItem CORBA interface and to supply GUI-specific functionality.

3.12.1.2 GUITrafficEventModuleUtilityClasses (Class Diagram)

This diagram shows the utility classes used in the GUITrafficEventModule, as well as the GUIResponseParticipation hierarchy, which would not fit on the GUITrafficEventClasses diagram.

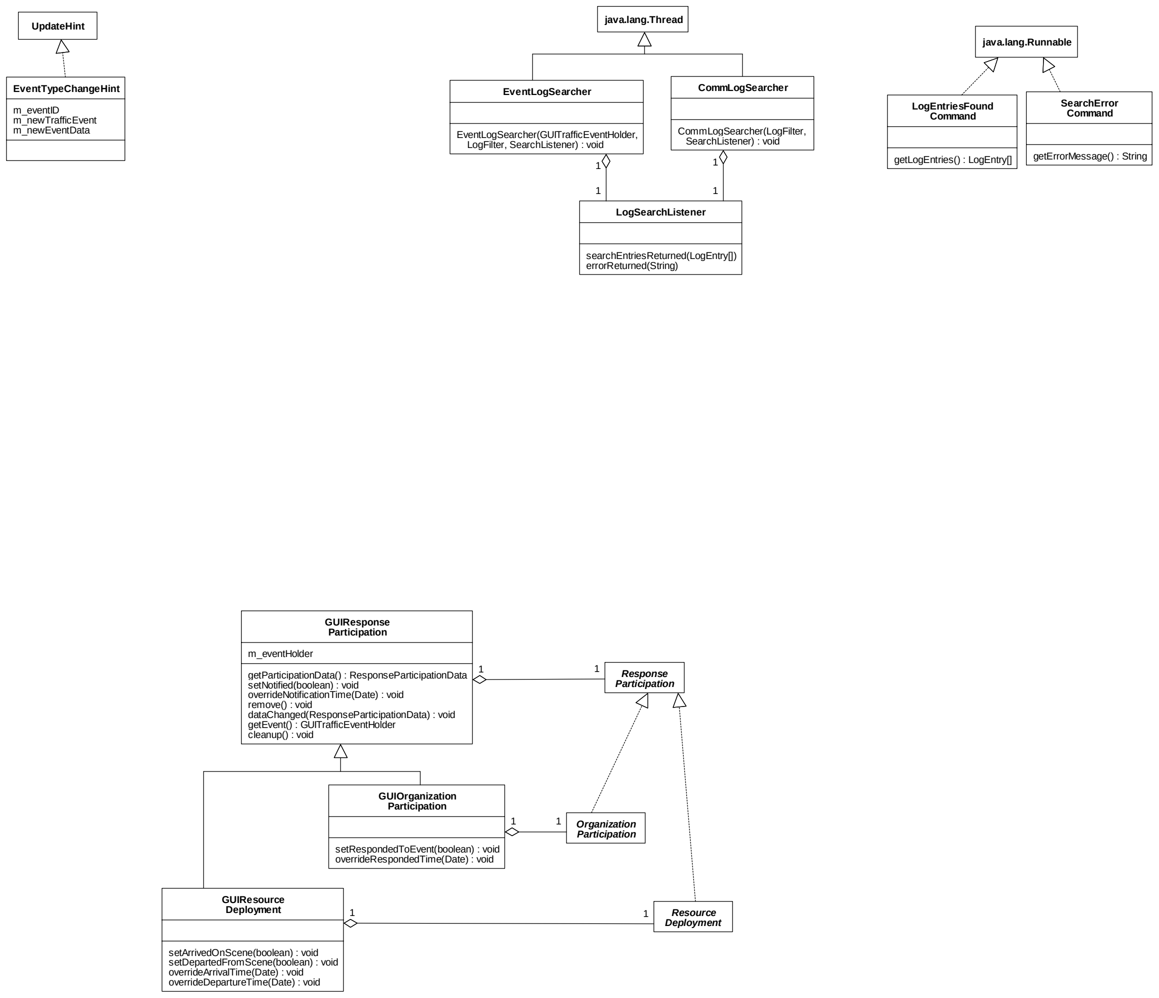

UpdateHint
EventTypeChangeHint
m_eventID
m_newTrafficEvent
m_newEventData
java.lang.Thread
EventLogSearcher
EventLogSearcher(GUITrafficEventHolder, LogFilter, SearchListener) : void
CommLogSearcher
CommLogSearcher(LogFilter, SearchListener) : void
LogSearchListener
searchEntriesReturned(LogEntry[])
errorReturned(String)
java.lang.Runnable
LogEntriesFound Command
getLogEntries() : LogEntry[]
SearchError Command
getErrorMessage() : String
GUIResponse Participation
m_eventHolder
getParticipationData() : ResponseParticipationData
setNotified(boolean) : void
overrideNotificationTime(Date) : void
remove() : void
dataChanged(ResponseParticipationData) : void
getEvent() : GUITrafficEventHolder
cleanup() : void
Response Participation
GUIOrganization Participation
setRespondedToEvent(boolean) : void
overrideRespondedTime(Date) : void
Organization Participation
GUIResource Deployment
setArrivedOnScene(boolean) : void
setDepartedFromScene(boolean) : void
overrideArrivalTime(Date) : void
overrideDepartureTime(Date) : void
Resource Deployment
1

Figure 136. GUITrafficEventModuleUtilityClasses (Class Diagram)

3.12.1.2.1 CommLogSearcher (Class)

This class provides functionality for starting an asynchronous search of the communications log.

3.12.1.2.2 EventLogSearcher (Class)

This class provides functionality for starting an asynchronous search of the traffic event's log.

3.12.1.2.3 EventTypeChangeHint (Class)

This hint object contains the information necessary to substitute a new TrafficEvent for an old TrafficEvent when the type of the traffic event changes.

3.12.1.2.4 GUIOrganization Participation (Class)

This GUI wrapper object wraps an OrganizationParticipation CORBA object and adds GUI-specific functionality. It represents an organization participating in the response to a traffic event.

3.12.1.2.5 GUIResource Deployment (Class)

This GUI wrapper object wraps a ResourceDeployment CORBA object and adds GUI-specific functionality. It represents an instance of a resource that has been deployed to respond to a traffic event.

3.12.1.2.6 GUIResponse Participation (Class)

This class represents one instance of a participant (person, mobile unit, device, resource, or response team) being notified and involved in the response to an event. This is a wrapper object which wraps a ResponseParticipation CORBA interface object, caches the CORBA object's data, and/or adds GUI-specific functionality.

3.12.1.2.7 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.12.1.2.8 java.lang.Thread (Class)

This class represents a java thread of execution.

3.12.1.2.9 LogSearchListener (Class)

This interface will allow the implementing class to receive log entries from an asynchronous log search as the search progresses. It also is used to report error messages if the search fails.

3.12.1.2.10 LogEntriesFound Command (Class)

This runnable is used to communicate that log entries have been found during an asynchronous search.

3.12.1.2.11 Organization Participation (Class)

This class is used to manage the data captured when an operator notifies another organization of a traffic event.

3.12.1.2.12 Resource Deployment (Class)

This class is used to store the data captured when an operator deploys resources to the scene of a traffic event.

3.12.1.2.13 Response Participation (Class)

This interface represents the involvement of one particular resource or organization in response to a particular traffic event.

3.12.1.2.14 SearchError Command (Class)

This Runnable is used to indicate that an error occurred while processing an asynchronous search.

3.12.1.2.15 UpdateHint (Class)

This interface must be implemented by all objects that are to be used as update hints. An update hint is a concept that is negotiated between a (subject) object and observers that are interested in that object. The data model makes no assumptions about how the hints will be used. The data model will invoke the isEqual method of the update hint to ask it to determine if it is equivalent to another hint. This allows the model to perform update optimizations by not sending notification to observers of two updates with equivalent hints in the same period. An example of how an update hint would be used follows: A DMS object has state variables that track the current message being displayed and the current latitude and longitude location of the sign controller. Because the system map requires significant processing load to redraw and needs only be notified if the latitude or longitude of the DMS changes the DMS and map view use a DMSMapChange hint. When the DMS object has a state change to the latitude or longitude property to report, that change is reported by calling objectUpdated and passing a DMSMapChange hint. When it has other changes which are not state changes to the latitude or longitude properties, it reports those changes to the DataModel by calling objectUpdated passing a DMSNonMapChange update hint. The map view will only redraw the DMS if the ObjectUpdate contains a DMSMapChange hint.

3.12.1.3 EventDialogs (Class Diagram)

This diagram shows the dialogs and GUI components that will be used to display the traffic event data, and it also shows which components are used for each type of traffic event.

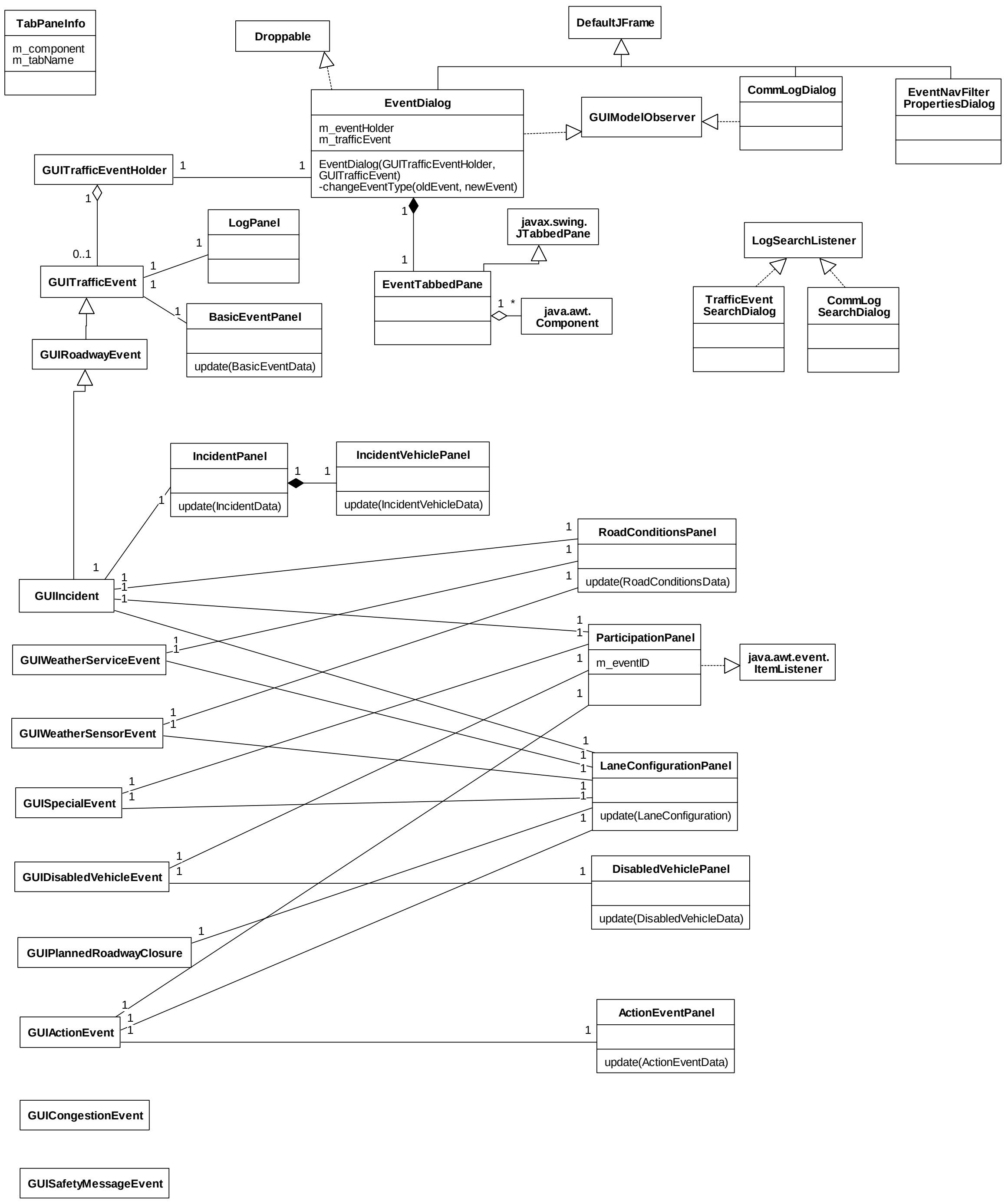


Figure 137. EventDialogs (Class Diagram)

3.12.1.3.1 ActionEventPanel (Class)

This JPanel will contain controls for specifying data specific to ActionEvents.

3.12.1.3.2 BasicEventPanel (Class)

This JPanel will contain the controls for entering data that is common to all traffic events.

3.12.1.3.3 DefaultJFrame (Class)

This class provides a default implementation of the WindowManageable interface, and may be used as a base class for other frame windows in the GUI. It handles all interactions with the WindowManager for attaching and detaching, as well as saving the window position.

3.12.1.3.4 DisabledVehiclePanel (Class)

This JPanel will contain controls for specifying data specific to DisabledVehicleEvents.

3.12.1.3.5 Droppable (Class)

This interface must be implemented by any object wishing to take part in a drag and drop operation. It is used by the DropHandler class to determine if a drop action should be allowed and to delegate the handling of the drop action after it is performed.

3.12.1.3.6 TabPaneInfo (Class)

This simple structure contains the information necessary to add a page to a JTabbedPane.

3.12.1.3.7 CommLog SearchDialog (Class)

This dialog allows the user to search the entries in the communications log for entries that fit the user defined search criteria.

3.12.1.3.8 CommLogDialog (Class)

This dialog is the GUI interface that allows the user to view, add, and search the entries in the communications log.

3.12.1.3.9 EventTabbedPane (Class)

This JTabbedPane will contain the panels specific to the type of traffic event.

3.12.1.3.10 javax.swing. JTabbedPane (Class)

This class is a component that has tabbed pages, and the user can click on a tab to flip to a certain page.

3.12.1.3.11 EventDialog (Class)

This is the main dialog for managing a traffic event. It will contain a tabbed pane that will allow the user to change information about the event or activate a response plan for the traffic event. The type of panels displayed in the tabbed pane depends on the type of event. If the type of the event is changed, the tabbed pane will be updated to display the correct panels for the new type of event.

3.12.1.3.12 GUIActionEvent (Class)

This is a wrapper class that wraps the ActionEvent CORBA interface. It will cache the ActionEvent data and supply any type-specific GUI functionality related to the TrafficEvent being an ActionEvent.

3.12.1.3.13 GUICongestionEvent (Class)

This is a wrapper class that wraps the CongestionEvent CORBA interface. It will cache the CongestionEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a CongestionEvent.

3.12.1.3.14 GUISafetyMessageEvent (Class)

This is a wrapper class that wraps the SafetyMessageEvent CORBA interface. It will cache the SafetyMessageEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a SafetyMessageEvent.

3.12.1.3.15 GUITrafficEvent (Class)

This class is a base class for the wrappers that wrap the TrafficEvent CORBA interface. The implementing object will exist from when the TrafficEvent is created until the GUI is shut down, the TrafficEvent type is changed, or the TrafficEvent is removed from the system. The class may cache the TrafficEvent data, and italso provides GUI functionality for the specific type of TrafficEvent.

3.12.1.3.16 GUIDisabledVehicleEvent (Class)

This is a wrapper class that wraps the DisabledVehicleEvent CORBA interface. It will cache the DisabledVehicleEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a DisabledVehicleEvent.

3.12.1.3.17 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

3.12.1.3.18 GUIPlannedRoadwayClosure (Class)

This is a wrapper class that wraps the PlannedRoadwayClosure CORBA interface. It will cache the PlannedRoadwayClosure data and supply any type-specific GUI functionality related to the TrafficEvent being a PlannedRoadwayClosure.

3.12.1.3.19 GUISpecialEvent (Class)

This is a wrapper class that wraps the SpecialEvent CORBA interface. It will cache the SpecialEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a SpecialEvent.

3.12.1.3.20 GUITrafficEventHolder (Class)

This object represents a TrafficEvent and provides GUI functionality for the TrafficEvent. This class contains generic data and operations that apply to any type of TrafficEvent. It also “holds” a type-specific GUITrafficEvent. If the type of the TrafficEvent is changed, the old GUITrafficEvent object (stored within this “holder” class) will be switched out for a new GUITrafficEvent of a different type, but the GUITrafficEventHolder will remain in existence.

3.12.1.3.21 IncidentVehiclePanel (Class)

This JPanel will contain controls for specifying the quantity and type of vehicles involved in an Incident.

3.12.1.3.22 LogPanel (Class)

This JPanel will contain the log component for the traffic event.

3.12.1.3.23 GUIWeatherSensorEvent (Class)

This is a wrapper class that wraps the WeatherSensorEvent CORBA interface. It will cache the WeatherSensorEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a WeatherSensorEvent.

3.12.1.3.24 GUIWeatherServiceEvent (Class)

This is a wrapper class that wraps the WeatherServiceEvent CORBA interface. It will cache the WeatherServiceEvent data and supply any type-specific GUI functionality related to the TrafficEvent being a WeatherServiceEvent.

3.12.1.3.25 ParticipationPanel (Class)

This JPanel will contain controls for specifying the state of the participations for any resources, mobile units, special needs teams, or external organizations that may be participating in the response to the traffic event.

3.12.1.3.26 EventNavFilter PropertiesDialog (Class)

This dialog allows the user to modify the properties of an EventNavFilter navigator filter.

3.12.1.3.27 GUIIncident (Class)

This is a wrapper class that wraps the Incident CORBA interface. It will cache the Incident data and supply any type-specific GUI functionality related to the TrafficEvent being an Incident.

3.12.1.3.28 IncidentPanel (Class)

This JPanel will contain controls for editing the data specific to Incident events.

3.12.1.3.29 GUIRoadwayEvent (Class)

This class extends the GUITrafficEvent class, adding any functionality that may be specific to the event being located on a roadway.

3.12.1.3.30 java.awt. Component (Class)

This class is the base class for all graphical user interface components such as buttons and panels.

3.12.1.3.31 java.awt.event. ItemListener (Class)

This interface allows the implementing class to listen for changes to an item such as a list item or combo box item.

3.12.1.3.32 LaneConfigurationPanel (Class)

This JPanel will contain controls for specifying the lane configuration and also the blockage state of each lane within that configuration.

3.12.1.3.33 LogSearchListener (Class)

This interface will allow the implementing class to receive log entries from an asynchronous log search as the search progresses. It also is used to report error messages if the search fails.

3.12.1.3.34 RoadConditionsPanel (Class)

This JPanel will contain controls for specifying the conditions of the road surface related to a traffic event.

3.12.1.3.35 TrafficEvent SearchDialog (Class)

This dialog allows the user to search the entries in the traffic event's log for entries that fit the user defined search criteria.

3.12.2 Sequence Diagrams

3.12.2.1 GUITrafficEventModule:AddCommLogEntry (Sequence Diagram)

This diagram shows how a user adds an entry to the Communications Log. The user invokes the Comm Log dialog from the GUI toolbar or by a hot key. The dialog attaches itself to the DataModel to be updated when any new entries are added, and queries the current entries from the GUICommLog wrapper. When the user enters a line in the Comm Log and hits the Enter key, the dialog calls the GUICommLog, which calls the CommLog CORBA interface to add the entry. An event will be pushed by the server through the Comm Log event channel and the GUI will be updated; otherwise, the error will be displayed in the dialog.

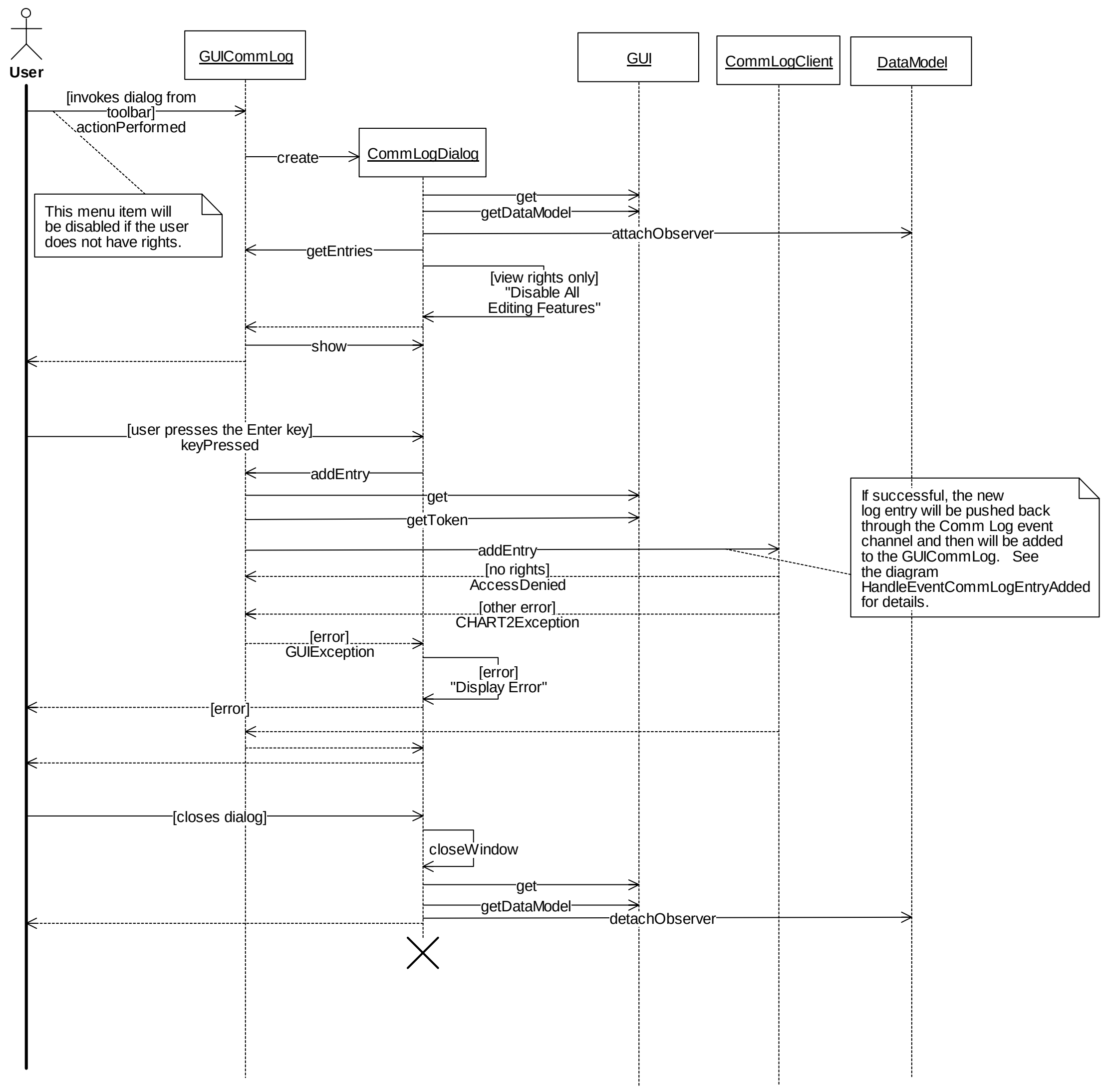
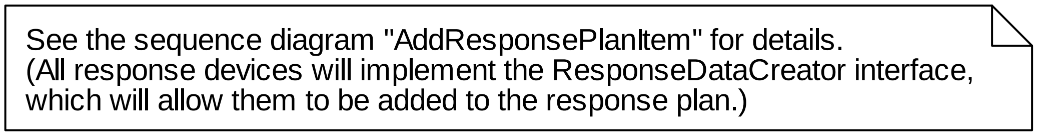

User
GUICommLog
GUI
CommLogClient
DataModel
[invokes dialog from toolbar] actionPerformed
create
CommLogDialog
This menu item will be disabled if the user does not have rights.
get
getDataModel
attachObserver
getEntries
[view rights only] "Disable All Editing Features"
show
[user presses the Enter key] keyPressed
addEntry
get
getToken
addEntry
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
If successful, the new log entry will be pushed back through the Comm Log event channel and then will be added to the GUICommLog. See the diagram HandleEventCommLogEntryAdded for details.
[closes dialog]
closeWindow
get
getDataModel
detachObserver

Figure 138. GUITrafficEventModule:AddCommLogEntry (Sequence Diagram)

3.12.2.2 GUITrafficEventModule:AddDeviceToResponse (Sequence Diagram)

This diagram shows how a device is added to a response plan. Any GUI wrapper of a response device must implement the ResponseDataCreator interface, which will be called to create a ResponsePlanItem for the device. See the sequence diagram: AddResponsePlanItem for details.



See the sequence diagram "AddResponsePlanItem" for details.
(All response devices will implement the ResponseDataCreator interface, which will allow them to be added to the response plan.)

Figure 139. GUITrafficEventModule:AddDeviceToResponse (Sequence Diagram)

3.12.2.3 GUITrafficEventModule:AddEvent (Sequence Diagram)

This diagram shows how a traffic event is added to the system from the GUI. The operator chooses “New Incident” (or another type of event) from the EventNavGroup’s context menu, or clicks on one of the buttons to create an event from the Comm Log dialog. (If the event is created from the Comm Log, the event will be initialized with any selected log entries). The EventNavGroup creates a GUITrafficEventHolder and a GUITrafficEvent object and calls doProperties() on the GUITrafficEventHolder to display the EventDialog. The dialog initializes the tab panes by calling the GUITrafficEvent, which passes back the correct tab panes corresponding to the specific type of the traffic event. It also gets the available traffic event factories, which will allow the user to choose which factory to use. The dialog is then displayed. As the user types in the dialog, the dialog will validate the input and enable the “Open Event” button when the required data has been entered for opening the traffic event (i.e., adding it to the System). When the user clicks on “Open Event”, the dialog calls the GUITrafficEvent to create the a specific type of BasicEventData object (depending on the type of GUITrafficEvent) and retrieves all of the data from the event-type-specific panels by passing the BasicEventData to the panels to have them fill in. The dialog will then call the GUITrafficEvent to open the event. It will try to call the selected TrafficEventFactory, or if none was selected, it will try to call each of the TrafficEventFactory objects until a TrafficEvent is successfully created. A TrafficEvent is returned synchronously from the factory and is set into the GUITrafficEvent wrapper. The GUITrafficEventHolder and GUITrafficEvent wrappers are then added to the DataModel so that the GUI can tell that the wrapper objects already exist when the corresponding CORBA event is pushed by the server and received by the GUI. (This allows the EventDialog to remain open because the GUI can retain the reference to the GUITrafficEventHolder object for future use and can therefore ignore the CORBA event; otherwise, a new GUITrafficEventHolder object would be created for the same traffic event when the CORBA event is handled.) For all other GUIs, the CORBA event will not be ignored and a new GUITrafficEventHolder will be created. See the sequence diagram: HandleEventEventAdded for more details on the handling of the CORBA event.

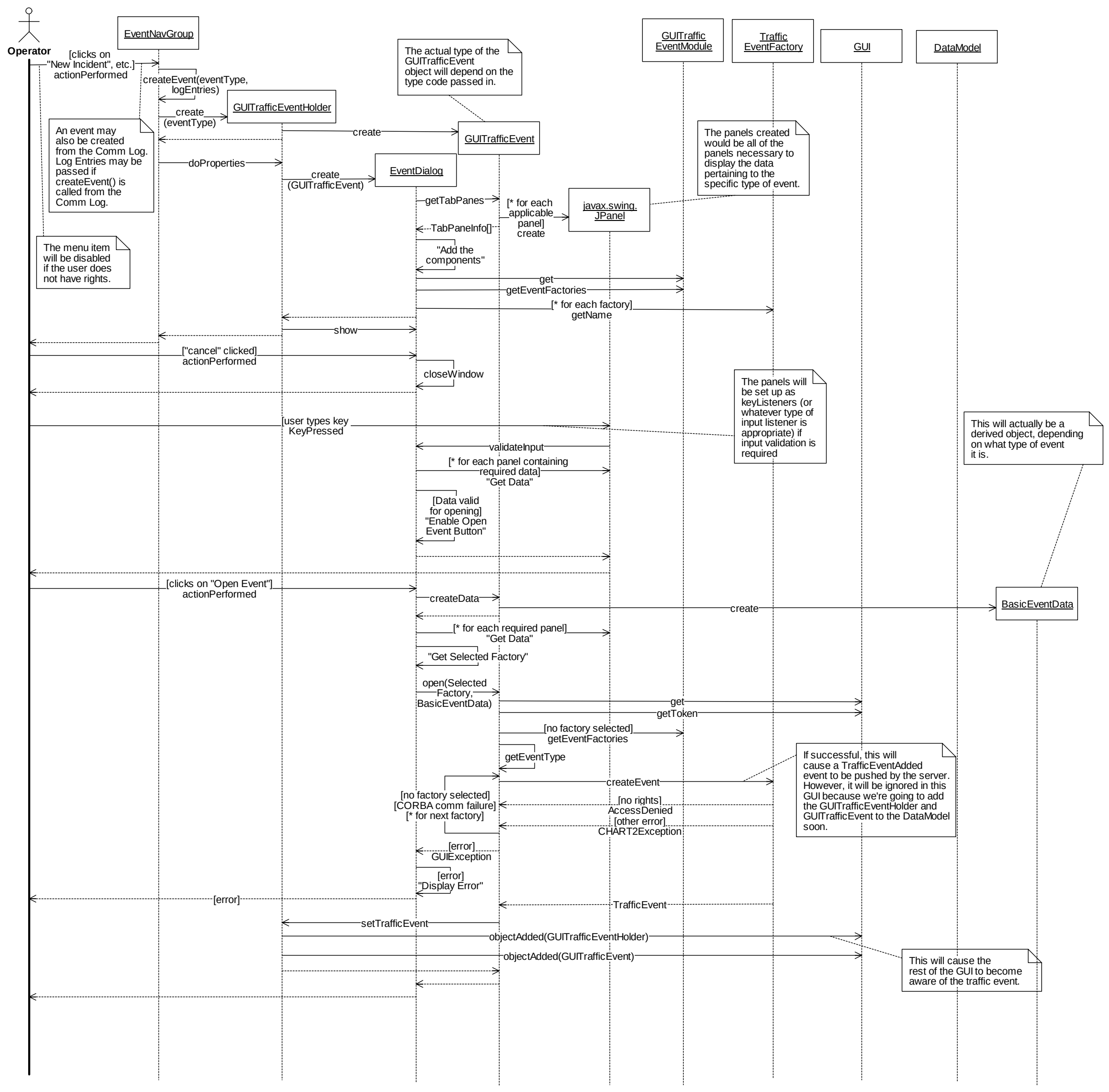
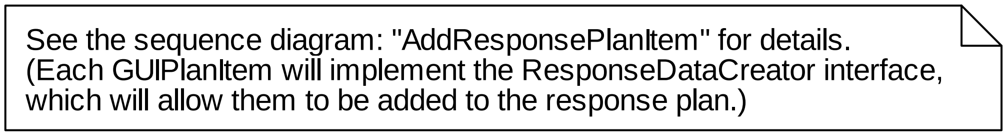

Operator
EventNavGroup
GUITraffic EventModule
Traffic EventFactory
GUI
DataModel
[clicks on "New Incident", etc.] actionPerformed
createEvent(eventType, logEntries)
The actual type of the GUITrafficEvent object will depend on the type code passed in.
create (eventType)
GUITrafficEventHolder
create
GUITrafficEvent
An event may also be created from the Comm Log. Log Entries may be passed if createEvent() is called from the Comm Log.
doProperties
create (GUITrafficEvent)
EventDialog
The panels created would be all of the panels necessary to display the data pertaining to the specific type of event.
getTabPanes
[* for each applicable panel] create
javax.swing. JPanel
TabPaneInfo[]
"Add the components"
The menu item will be disabled if the user does not have rights.
get
getEventFactories
[* for each factory] getName
show
["cancel" clicked] actionPerformed
closeWindow
The panels will be set up as keyListeners (or whatever type of input listener is appropriate) if input validation is required
This will actually be a derived object, depending on what type of event it is.
[user types key KeyPressed
validateInput
[* for each panel containing required data] "Get Data"
[Data valid for opening] "Enable Open Event Button"
[clicks on "Open Event"] actionPerformed
createData
create
BasicEventData
[* for each required panel] "Get Data"
"Get Selected Factory"
open(Selected Factory, BasicEventData)
get
getToken
[no factory selected] getEventFactories
getEventType
If successful, this will cause a TrafficEventAdded event to be pushed by the server. However, it will be ignored in this GUI because we're going to add the GUITrafficEventHolder and GUITrafficEvent to the DataModel soon.
createEvent
[no factory selected] [CORBA comm failure] [* for next factory]
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
TrafficEvent
setTrafficEvent
objectAdded(GUITrafficEventHolder)
objectAdded(GUITrafficEvent)
This will cause the rest of the GUI to become aware of the traffic event.

Figure 140. GUITrafficEventModule:AddEvent (Sequence Diagram)

3.12.2.4 GUITrafficEventModule:AddPlanItemToResponse (Sequence Diagram)

This diagram shows how a plan item is added to a response plan. The GUIPlanItem objects will implement the ResponseDataCreator interface, which will be called to create a ResponsePlanItem for the plan item. See the sequence diagram: AddResponsePlanItem for details.

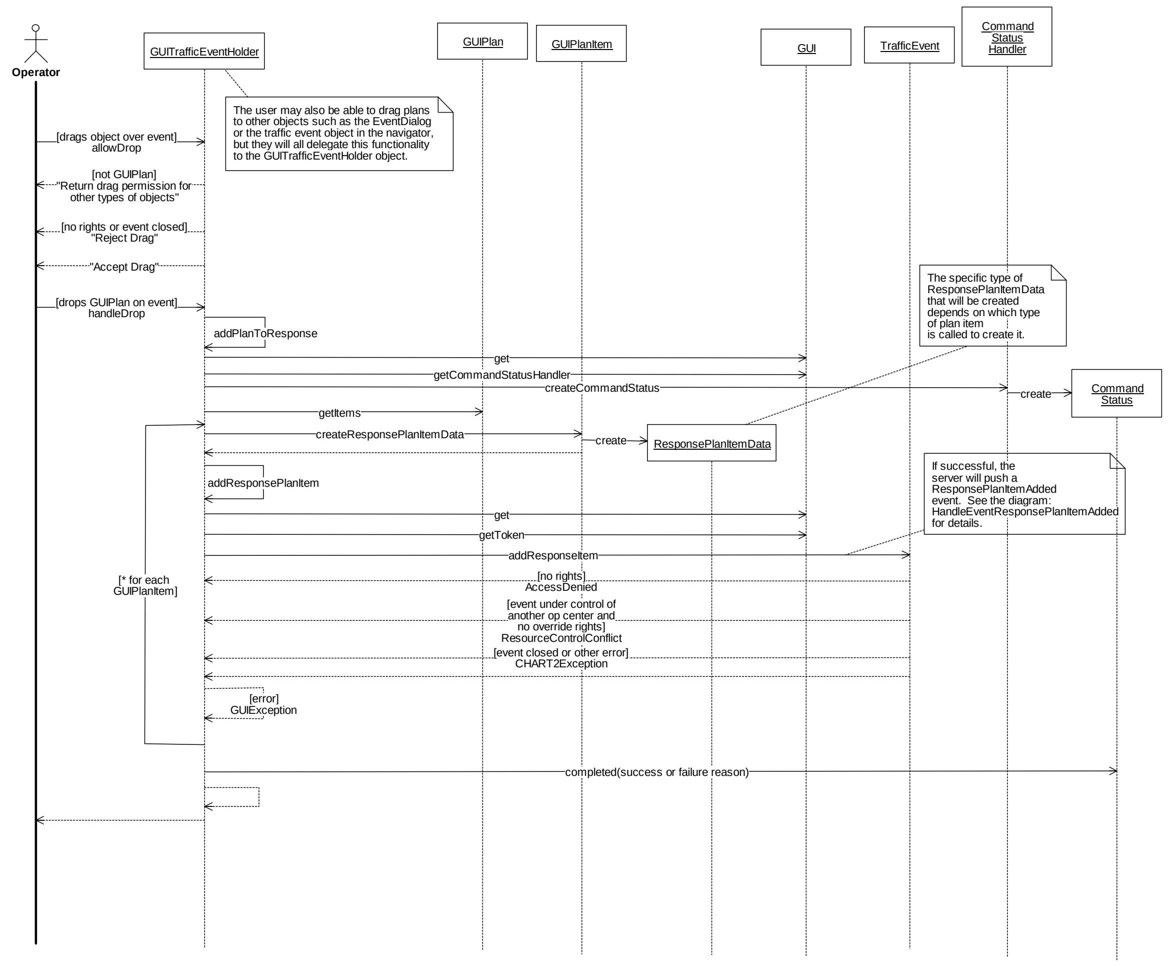


See the sequence diagram: "AddResponsePlanItem" for details.
(Each GUIPlanItem will implement the ResponseDataCreator interface, which will allow them to be added to the response plan.)

Figure 141. GUITrafficEventModule:AddPlanItemToResponse (Sequence Diagram)

3.12.2.5 GUITrafficEventModule:AddPlanToResponse (Sequence Diagram)

This diagram shows how a Plan is added to a Response Plan for an event. The user drags and drops the GUIPlan object onto the GUITrafficEventHolder object. The GUITrafficEventHolder then gets the items from the GUIPlan and for each one, calls it to create its specific type of ResponsePlanItemData for its specific type of plan item. Then the GUITrafficEventHolder calls the TrafficEvent to add the response plan item. If successful, the server will push a CORBA event through the TrafficEvent channel. See the sequence diagram: HandleEventResponsePlanItemAdded for details of the handling of this event. If an error occurs, a CommandStatus object will be created so that the failure will appear in the Command Failures window.



Operator
GUITrafficEventHolder
GUIPlan
GUIPlanItem
GUI
TrafficEvent
Command Status Handler
The user may also be able to drag plans to other objects such as the EventDialog or the traffic event object in the navigator, but they will all delegate this functionality to the GUITrafficEventHolder object.
[drags object over event] allowDrop
[not GUIPlan] "Return drag permission for other types of objects"
[no rights or event closed] "Reject Drag"
"Accept Drag"
The specific type of ResponsePlanItemData that will be created depends on which type of plan item is called to create it.
[drops GUIPlan on event] handleDrop
addPlanToResponse
get
getCommandStatusHandler
createCommandStatus
create
Command Status
getItems
createResponsePlanItemData
create
ResponsePlanItemData
If successful, the server will push a ResponsePlanItemAdded event. See the diagram: HandleEventResponsePlanItemAdded for details.
addResponsePlanItem
get
getToken
addResponseItem
[* for each GUIPlanItem]
[no rights] AccessDenied
[event under control of another op center and no override rights] ResourceControlConflict
[event closed or other error] CHART2Exception
[error] GUIException
completed(success or failure reason)

Figure 142. GUITrafficEventModule:AddPlanToResponse (Sequence Diagram)

3.12.2.6 GUITrafficEventModule:AddResponseParticipation (Sequence Diagram)

This diagram shows how the user records a participation in reponse to an event. When the EventDialog is initialized, the ParticipationPanel will get the valid participant types from the GUITrafficEventHolder and will get the ResponseParticipant objects from the GUIOperationsCenter corresponding to the operations center where the user is logged in. Each participant of a type applicable to the current GUITrafficEvent will be added to the combo box. When the user chooses a response participant to participate in the event, the ParticipationPanel will call the GUITrafficEventHolder to create a ResponseParticipationData object (which will actually be a derived type depending on the type of participant that was chosen). The GUITrafficEventHolder will then call the TrafficEvent to add the response participation on the server side. If the participation is successfully added, the server will push out an event and the participation will be added as a row in the ParticipationPanel. See the sequence diagram: HandleEventResponseParticipationAdded for details. If the user types in a mobile unit that is not one of the units at the operations center, the ParticipationPanel will first check that mobile units are supported by the event type. If so, it will call the GUIOperationsCenter corresponding to the operations center where the user is logged in and ask it to add a response participant. If this is successful, the ParticipationPanel adds the participation as described above. An event will also be pushed by the server if a new participant is added to the OperationsCenter, so that other GUIs will also see the new participant.

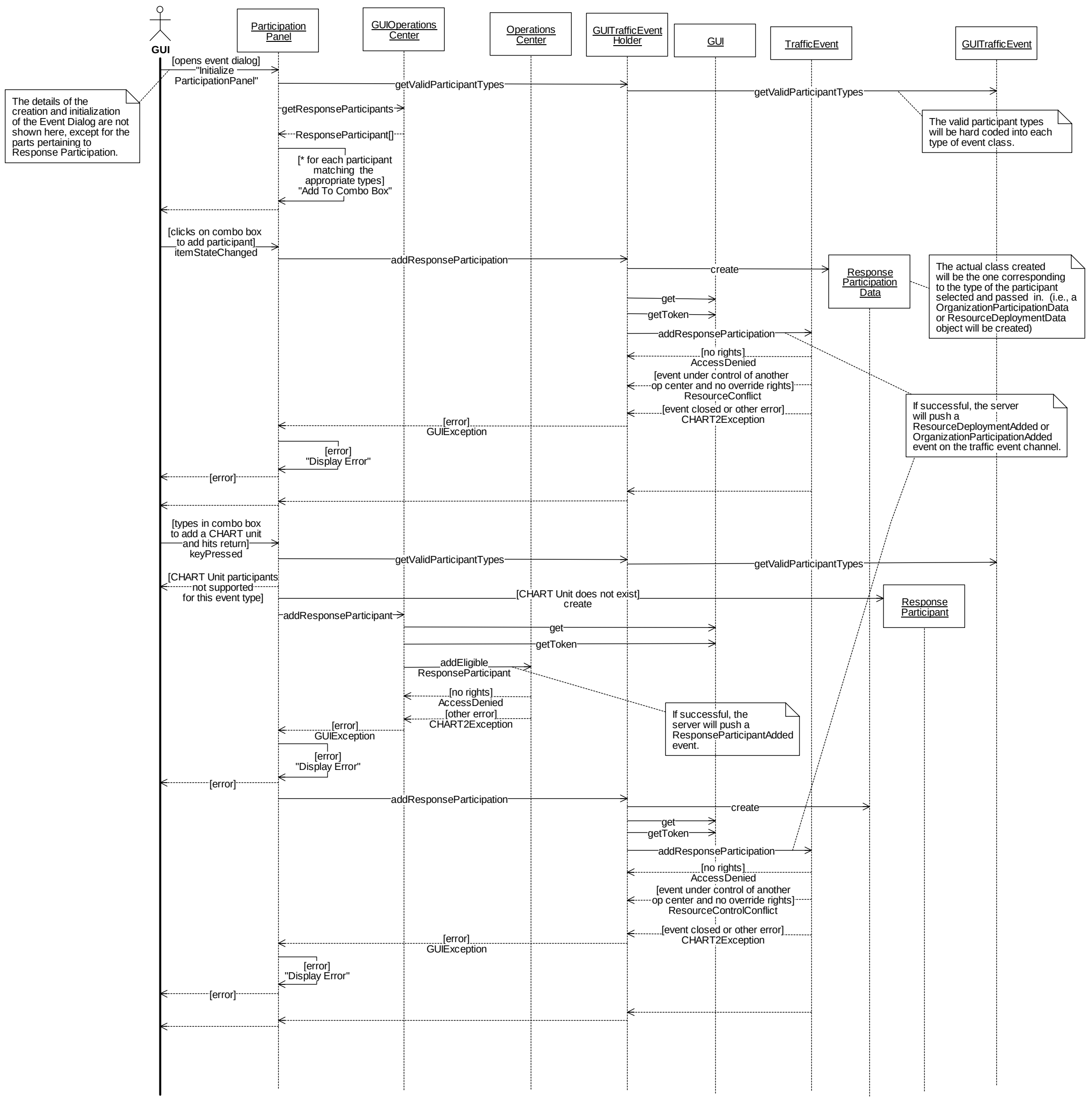
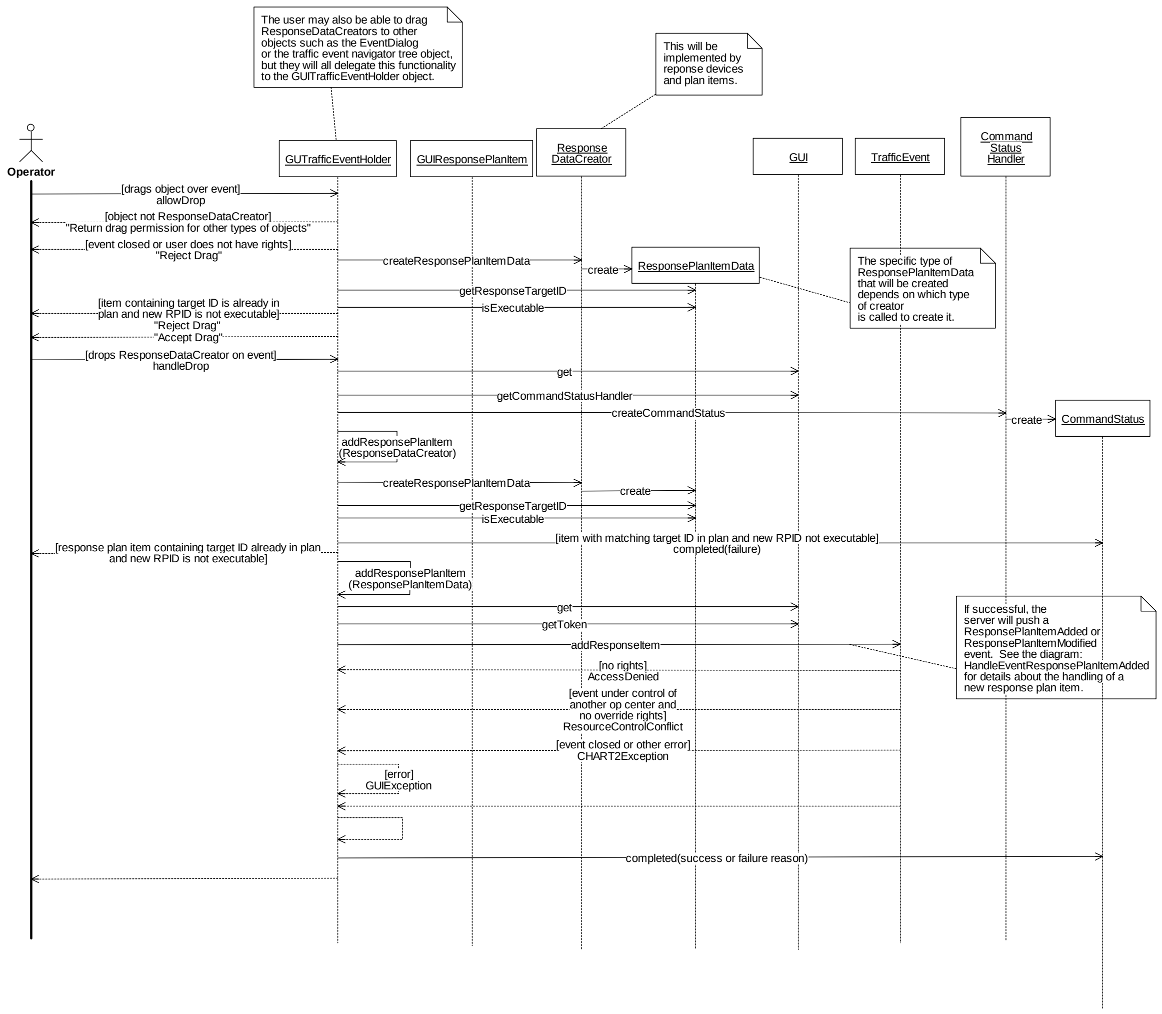

GUI
Participation Panel
GUIOperations Center
Operations Center
GUITrafficEvent Holder
GUI
TrafficEvent
GUITrafficEvent
[opens event dialog] "Initialize ParticipationPanel"
The details of the creation and initialization of the Event Dialog are not shown here, except for the parts pertaining to Response Participation.
getValidParticipantTypes
getValidParticipantTypes
The valid participant types will be hard coded into each type of event class.
getResponseParticipants
ResponseParticipant[]
[* for each participant matching the appropriate types] "Add To Combo Box"
[clicks on combo box to add participant] itemStateChanged
addResponseParticipation
create
Response Participation Data
The actual class created will be the one corresponding to the type of the participant selected and passed in. (i.e., a OrganizationParticipationData or ResourceDeploymentData object will be created)
get
getToken
addResponseParticipation
[no rights] AccessDenied
[event under control of another op center and no override rights] ResourceConflict
[event closed or other error] CHART2Exception
If successful, the server will push a ResourceDeploymentAdded or OrganizationParticipationAdded event on the traffic event channel.
[error] GUIException
[error] "Display Error"
[error]
[types in combo box to add a CHART unit and hits return] keyPressed
getValidParticipantTypes
getValidParticipantTypes
[CHART Unit participants not supported for this event type]
[CHART Unit does not exist] create
Response Participant
addResponseParticipant
get
getToken
addEligible ResponseParticipant
[no rights] AccessDenied
[other error] CHART2Exception
If successful, the server will push a ResponseParticipantAdded event.
[error] GUIException
[error] "Display Error"
[error]
addResponseParticipation
create
get
getToken
addResponseParticipation
[no rights] AccessDenied
[event under control of another op center and no override rights] ResourceControlConflict
[event closed or other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]

Figure 143. GUITrafficEventModule:AddResponseParticipation (Sequence Diagram)

3.12.2.7 GUITrafficEventModule:AddResponsePlanItem (Sequence Diagram)

This diagram shows how a Response Plan Item is added to an event's Response Plan. The user drags and drops a ResponseDataCreator object onto the GUITrafficEventHolder object. The GUITrafficEventHolder then calls it to create its specific type of ResponsePlanItemData for the specific type of ResponseDataCreator. Then the GUITrafficEventHolder calls the TrafficEvent to add the response plan item. If successful, the server will push a CORBA event through the TrafficEvent channel. See the sequence diagram: HandleEventResponsePlanItemAdded for details of the handling of this event. If an error occurs, a CommandStatus object will be created so that the failure will appear in the Command Failures window.



The user may also be able to drag ResponseDataCreators to other objects such as the EventDialog or the traffic event navigator tree object, but they will all delegate this functionality to the GUITrafficEventHolder object.
This will be implemented by reponse devices and plan items.
Operator
GUTrafficEventHolder
GUIResponsePlanItem
Response DataCreator
GUI
TrafficEvent
Command Status Handler
[drags object over event] allowDrop
[object not ResponseDataCreator] "Return drag permission for other types of objects"
[event closed or user does not have rights] "Reject Drag"
createResponsePlanItemData
create
ResponsePlanItemData
The specific type of ResponsePlanItemData that will be created depends on which type of creator is called to create it.
getResponseTargetID
isExecutable
[item containing target ID is already in plan and new RPID is not executable] "Reject Drag"
"Accept Drag"
[drops ResponseDataCreator on event] handleDrop
get
getCommandStatusHandler
createCommandStatus
create
CommandStatus
addResponsePlanItem (ResponseDataCreator)
createResponsePlanItemData
create
getResponseTargetID
isExecutable
[response plan item containing target ID already in plan and new RPID is not executable]
[item with matching target ID in plan and new RPID not executable] completed(failure)
addResponsePlanItem (ResponsePlanItemData)
get
getToken
addResponseItem
If successful, the server will push a ResponsePlanItemAdded or ResponsePlanItemModified event. See the diagram: HandleEventResponsePlanItemAdded for details about the handling of a new response plan item.
[no rights] AccessDenied
[event under control of another op center and no override rights] ResourceControlConflict
[event closed or other error] CHART2Exception
[error] GUIException
completed(success or failure reason)

Figure 144. GUITrafficEventModule:AddResponsePlanItem (Sequence Diagram)

3.12.2.8 GUITrafficEventModule:AddTextToEventHistory (Sequence Diagram)

This diagram shows how a text entry is added to the traffic event's history log. When a user types in a new entry and hits Enter, the Event Dialog calls the GUITrafficEventHolder to add a log entry. This in turn calls the TrafficEvent CORBA interface to add the log entry. If there is an error adding the entry, the error will be displayed in the dialog; otherwise, a CORBA event will be pushed by the server containing the new log entry, and the GUI will catch this event and add the row to the traffic event history display.

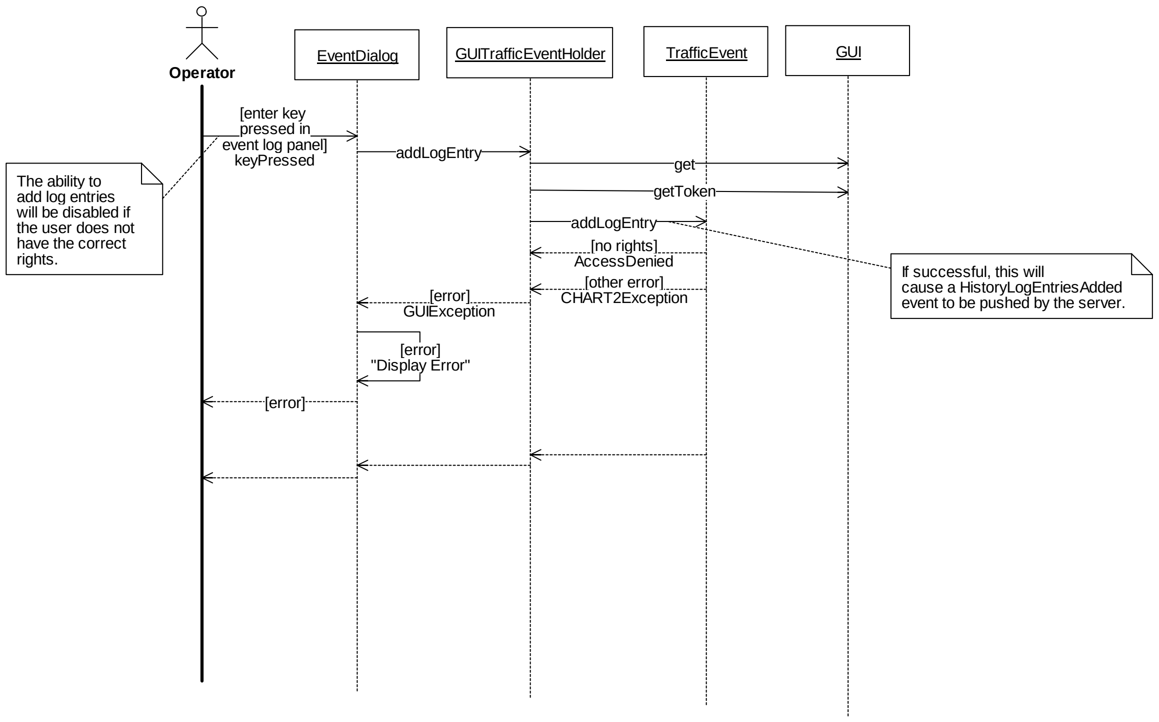

Operator
EventDialog
GUITrafficEventHolder
TrafficEvent
GUI
[enter key pressed in event log panel] keyPressed
addLogEntry
get
getToken
addLogEntry
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
The ability to add log entries will be disabled if the user does not have the correct rights.
If successful, this will cause a HistoryLogEntriesAdded event to be pushed by the server.

Figure 145. GUITrafficEventModule:AddTextToEventHistory (Sequence Diagram)

3.12.2.9 GUITrafficEventModule:AssociateEvent (Sequence Diagram)

This diagram shows how an event is associated from the GUI. The user drags the secondary event onto the primary event in the Navigator and drops it. The primary GUITrafficEventHolder will call its own associateEvent() method, passing the secondary event's GUI wrapper. The primary GUITrafficEventHolder will then call the secondary one, this time passing its own TrafficEvent CORBA interface object. The secondary GUITrafficEventHolder will then call the passed TrafficEvent object, passing its own TrafficEvent object as the event to associate. If successful, the server will push a CORBA event with the new association. See the sequence diagram: HandleEventEventAssociated for details. If the association fails, a CommandStatus object will be created so that the failure will appear in the Command Failures window.

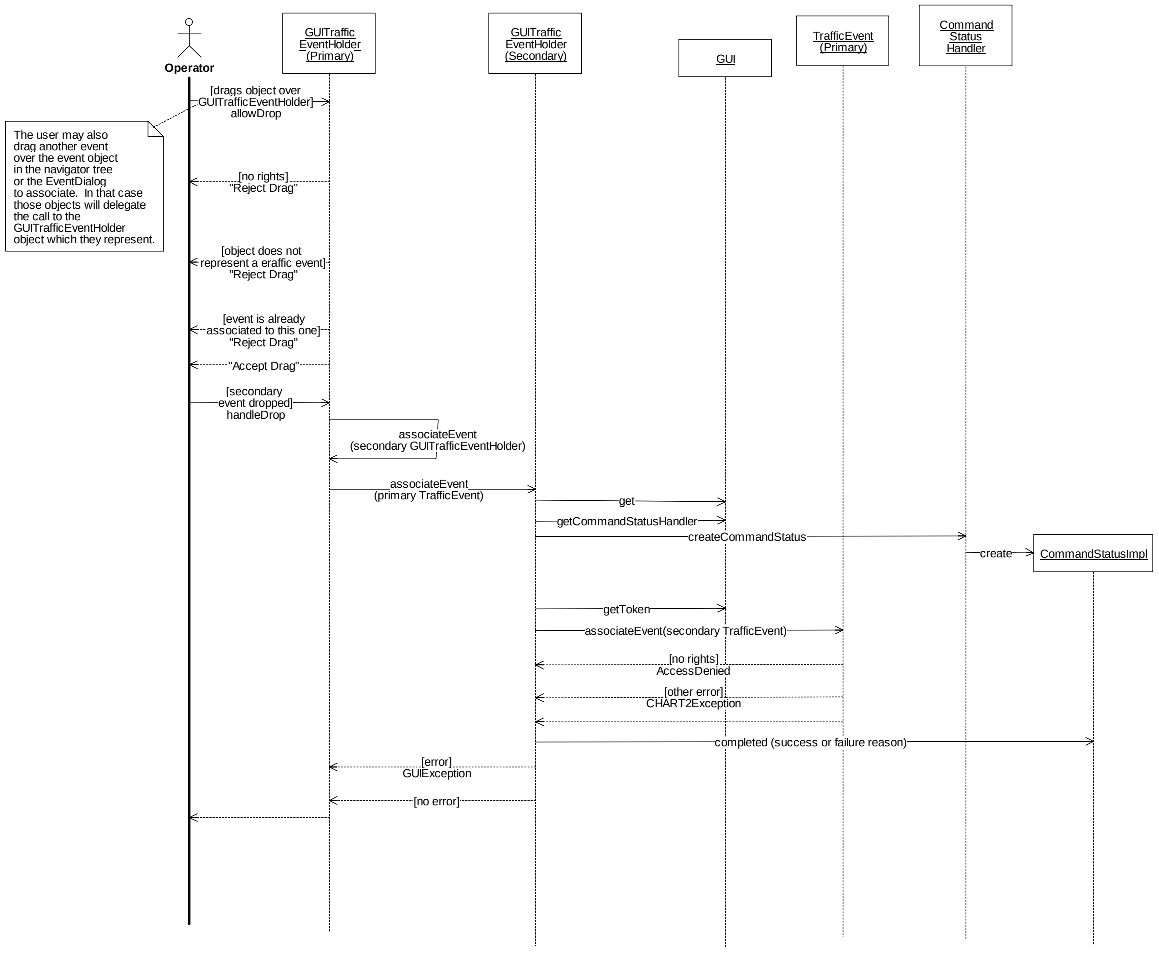
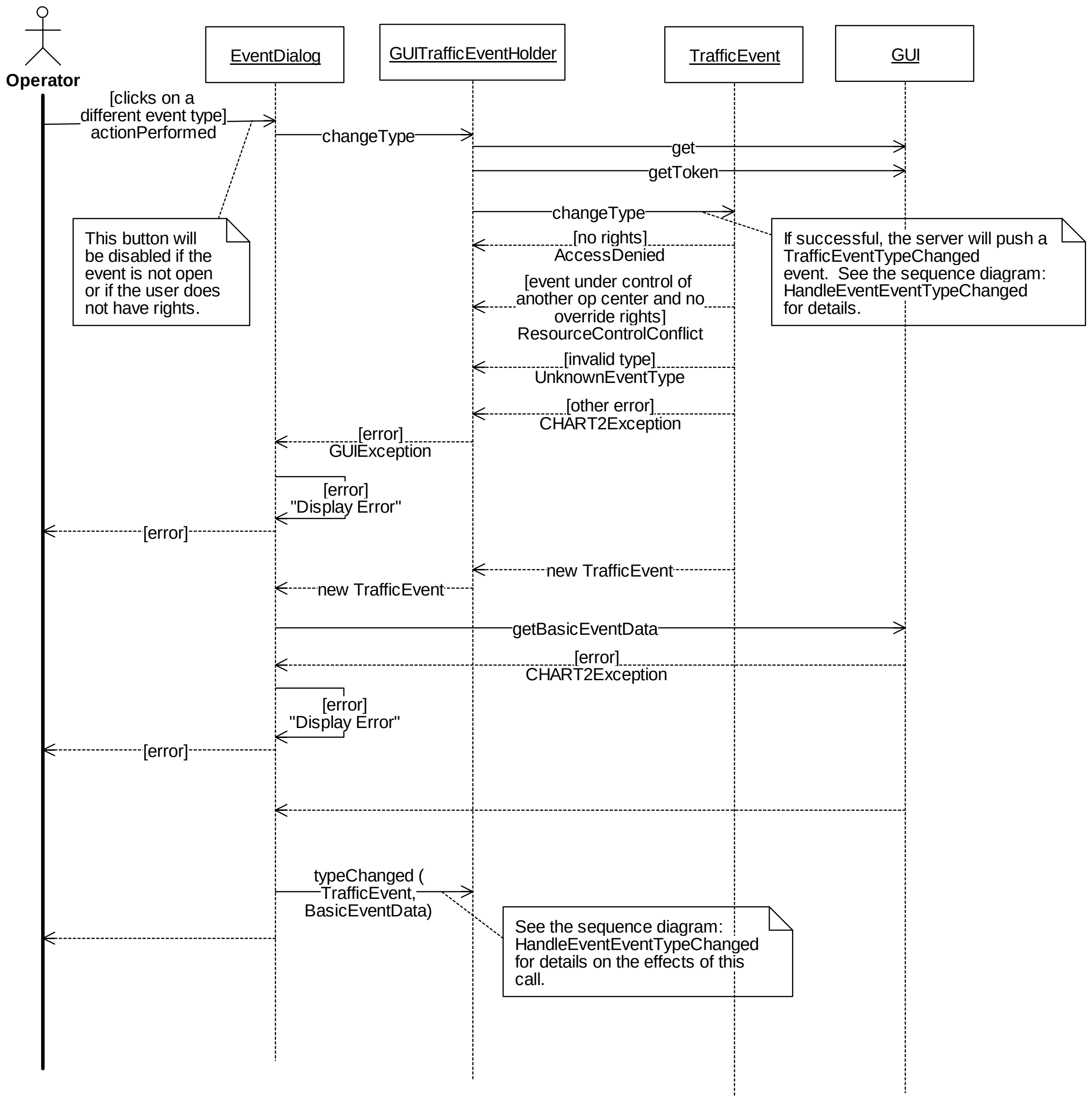

Operator
GUITraffic EventHolder (Primary)
GUITraffic EventHolder (Secondary)
GUI
TrafficEvent (Primary)
Command Status Handler
The user may also drag another event over the event object in the navigator tree or the EventDialog to associate. In that case those objects will delegate the call to the GUITrafficEventHolder object which they represent.
[drags object over GUITrafficEventHolder] allowDrop
[no rights] "Reject Drag"
[object does not represent a eraffic event] "Reject Drag"
[event is already associated to this one] "Reject Drag"
"Accept Drag"
[secondary event dropped] handleDrop
associateEvent (secondary GUITrafficEventHolder)
associateEvent (primary TrafficEvent)
get
getCommandStatusHandler
createCommandStatus
create
CommandStatusImpl
getToken
associateEvent(secondary TrafficEvent)
[no rights] AccessDenied
[other error] CHART2Exception
completed (success or failure reason)
[error] GUIException
[no error]

Figure 146. GUITrafficEventModule:AssociateEvent (Sequence Diagram)

3.12.2.10 GUITrafficEventModule:ChangeEventType (Sequence Diagram)

This diagram shows how the event type is changed. The user clicks on a button in the EventDialog indicating the type of event to change to. The EventDialog then calls the GUITrafficEventHolder wrapper, which in turn calls the TrafficEvent CORBA interface to change the event. If successful, the dialog will be changed immediately and a CORBA event will be pushed by the server to update all of the other GUIs; otherwise, an error message will be displayed in the dialog. See the sequence diagram: HandleEventEventTypeChanged for details on the handling of the CORBA event that is pushed.



Operator
EventDialog
GUITrafficEventHolder
TrafficEvent
GUI
[clicks on a different event type] actionPerformed
changeType
get
getToken
changeType
This button will be disabled if the event is not open or if the user does not have rights.
If successful, the server will push a TrafficEventTypeChanged event. See the sequence diagram: HandleEventEventTypeChanged for details.
[no rights] AccessDenied
[event under control of another op center and no override rights] ResourceControlConflict
[invalid type] UnknownEventType
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
new TrafficEvent
new TrafficEvent
getBasicEventData
[error] CHART2Exception
[error] "Display Error"
[error]
typeChanged (TrafficEvent, BasicEventData)
See the sequence diagram: HandleEventEventTypeChanged for details on the effects of this call.

Figure 147. GUITrafficEventModule:ChangeEventType (Sequence Diagram)

[DCE:293]

3.12.2.11 GUITrafficEventModule:CloseEvent (Sequence Diagram)

This diagram shows how a traffic event is closed from the GUI. The operator clicks on “close” from the GUITrafficEventHolder’s menu or from the EventDialog, and the GUITrafficEventHolder’s close() method is called, which calls the TrafficEvent CORBA interface. If the traffic event is closed, a CORBA event will be pushed. See the sequence diagram: HandleEventEventClosed for details.

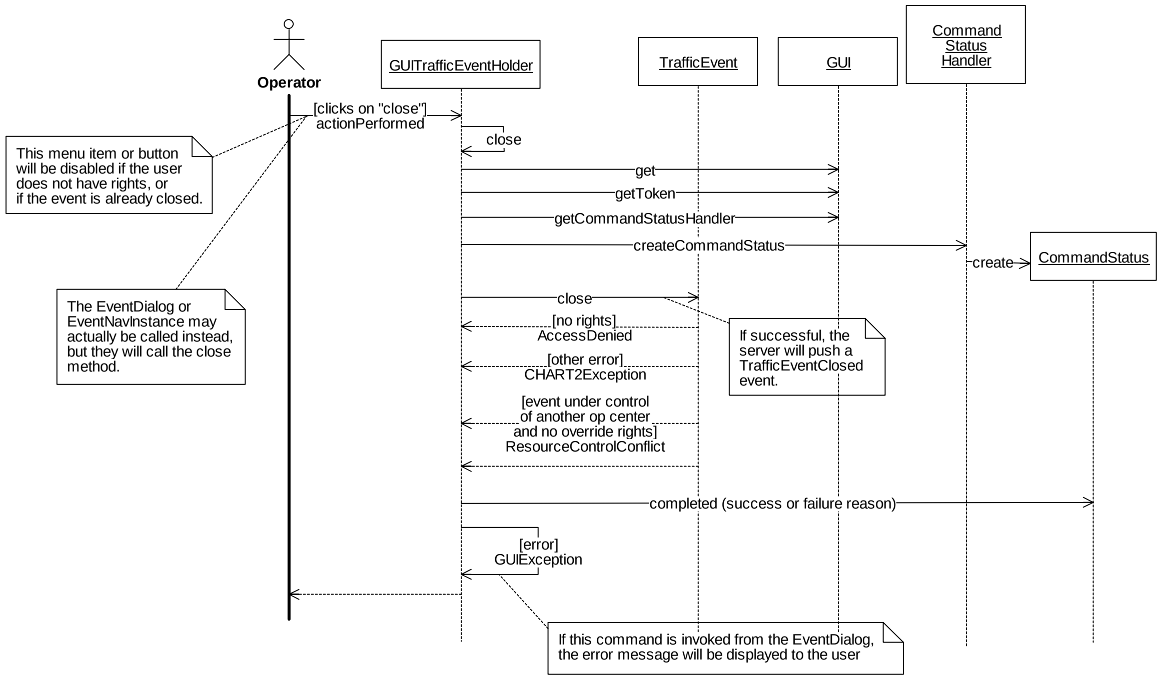

Operator
GUITrafficEventHolder
TrafficEvent
GUI
Command Status Handler
[clicks on "close"] actionPerformed
close
This menu item or button will be disabled if the user does not have rights, or if the event is already closed.
get
getToken
getCommandStatusHandler
createCommandStatus
create
CommandStatus
The EventDialog or EventNavInstance may actually be called instead, but they will call the close method.
close
[no rights] AccessDenied
[other error] CHART2Exception
[event under control of another op center and no override rights] ResourceControlConflict
If successful, the server will push a TrafficEventClosed event.
completed (success or failure reason)
[error] GUIException
If this command is invoked from the EventDialog, the error message will be displayed to the user

Figure 148. GUITrafficEventModule:CloseEvent (Sequence Diagram)

3.12.2.12 GUITrafficEventModule:Discovery (Sequence Diagram)

This diagram shows what happens in the GUITrafficEventModule during the discovery phase, when CORBA event channels and objects are discovered. First, the CORBA event channels are queried from the trader for TrafficEvent and CommLog event channels. PushEventConsumer objects are created for each channel and added to the EventConsumerGroup to maintain the connection to the event channels. During the object discovery phase, the module queries the CommLog objects from the trader and adds them to the GUICommLog object. Then the module queries the TrafficEventFactory objects, and gets the TrafficEvent objects from the factories. For each TrafficEvent, a GUITrafficEventHolder will be created and added to the DataModel if one does not already exist for that traffic event. A GUITrafficEvent object will be created, whose type depends on the type of traffic event discovered. Then the ResponseParticipation objects are obtained from the traffic event and added to the GUITrafficEventHolder and to the DataModel. Then traffic event associations are obtained from the TrafficEvent and TrafficEventAssociation objects are created and added to the DataModel. Then the ResponsePlanItems are retrieved, and for each one, the GUIResponsePlanItemCreators are called until a type-specific GUIResponsePlanItem wrapper object is created for the ResponsePlanItem. The GUIResponsePlanItems are added to the GUITrafficEventHolder and to the DataModel. Then the entries are retrieved from the TrafficEvent’s history log and are added to the GUITrafficEventHolder’s cache of log entries.

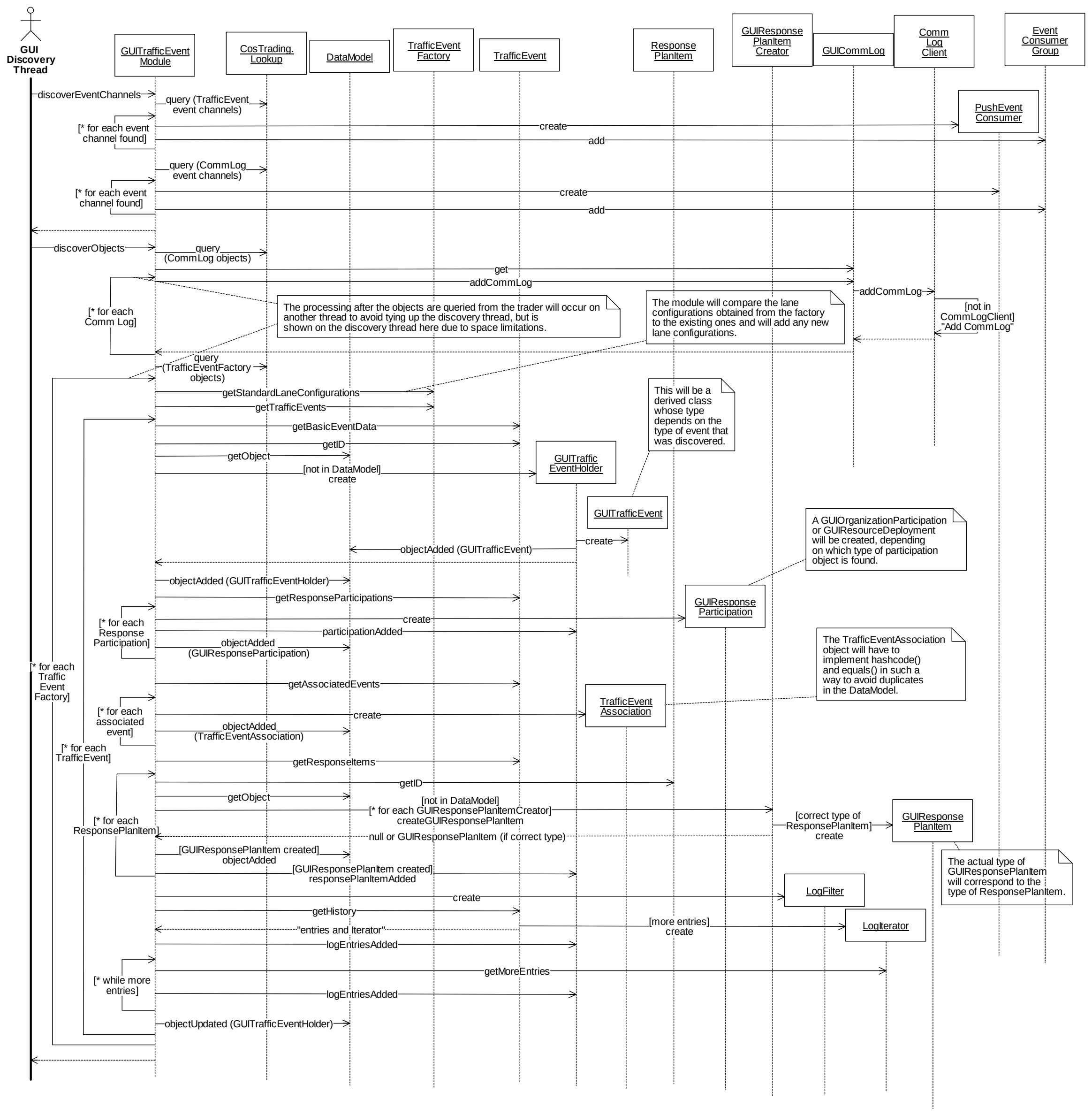
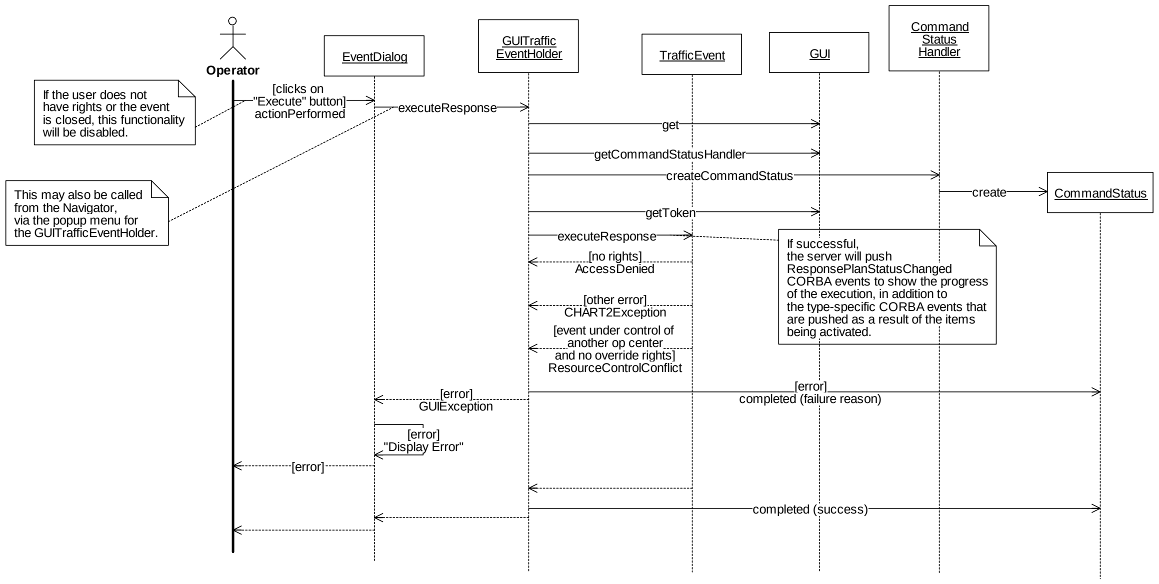

GUI Discovery Thread
GUITrafficEventModule
CosTrading.Lookup
DataModel
TrafficEventFactory
TrafficEvent
ResponsePlanItem
GUIResponsePlanItemCreator
GUICommLog
CommLogClient
EventConsumerGroup
PushEventConsumer
discoverEventChannels
query (TrafficEvent event channels)
[* for each event channel found]
create
add
query (CommLog event channels)
discoverObjects
query (CommLog objects)
get
addCommLog
[* for each Comm Log]
[not in CommLogClient] "Add CommLog"
The processing after the objects are queried from the trader will occur on another thread to avoid tying up the discovery thread, but is shown on the discovery thread here due to space limitations.
The module will compare the lane configurations obtained from the factory to the existing ones and will add any new lane configurations.
query (TrafficEventFactory objects)
getStandardLaneConfigurations
getTrafficEvents
getBasicEventData
getID
getObject
[not in DataModel] create
GUITrafficEventHolder
This will be a derived class whose type depends on the type of event that was discovered.
GUITrafficEvent
objectAdded (GUITrafficEvent)
objectAdded (GUITrafficEventHolder)
getResponseParticipations
A GUIOrganizationParticipation or GUIResourceDeployment will be created, depending on which type of participation object is found.
GUIResponseParticipation
[* for each Response Participation]
participationAdded
objectAdded (GUIResponseParticipation)
[* for each Traffic Event Factory]
getAssociatedEvents
The TrafficEventAssociation object will have to implement hashcode() and equals() in such a way to avoid duplicates in the DataModel.
TrafficEventAssociation
[* for each associated event]
objectAdded (TrafficEventAssociation)
[* for each TrafficEvent]
getResponseItems
[not in DataModel] [* for each GUIResponsePlanItemCreator] createGUIResponsePlanItem
[correct type of ResponsePlanItem] create
GUIResponsePlanItem
[* for each ResponsePlanItem]
null or GUIResponsePlanItem (if correct type)
[GUIResponsePlanItem created] objectAdded
[GUIResponsePlanItem created] responsePlanItemAdded
The actual type of GUIResponsePlanItem will correspond to the type of ResponsePlanItem.
LogFilter
getHistory
"entries and Iterator"
[more entries] create
LogIterator
logEntriesAdded
[* while more entries]
getMoreEntries
objectUpdated (GUITrafficEventHolder)

Figure 149. GUITrafficEventModule:Discovery (Sequence Diagram)

3.12.2.13 GUITrafficEventModule:ExecuteResponse (Sequence Diagram)

This diagram shows how a response plan is executed. The user clicks on the “Execute” button or menu item, and the GUITrafficEventHolder’s executeResponse() method is called, which calls the TrafficEvent’s executeReponse(). As the response plan is executed, the server will push an event to indicate that the response plan’s status has changed. If an error occurs immediately, a message will be displayed in the dialog or a CommandStatusImpl object will be created to show the command failure. If the response plan is successfully executed, the server will push a PlanStatusChanged event and the GUIResponsePlanItems will be updated to show the status of the individual items. Upon failure of individual items, a CommandStatusImpl object will be created to show the failure in the CommandFailures window.



Operator
EventDialog
GUITraffic EventHolder
TrafficEvent
GUI
Command Status Handler
CommandStatus
If the user does not have rights or the event is closed, this functionality will be disabled.
This may also be called from the Navigator, via the popup menu for the GUITrafficEventHolder.
[clicks on "Execute" button] actionPerformed
executeResponse
get
getCommandStatusHandler
createCommandStatus
create
getToken
executeResponse
If successful, the server will push ResponsePlanStatusChanged CORBA events to show the progress of the execution, in addition to the type-specific CORBA events that are pushed as a result of the items being activated.
[no rights] AccessDenied
[other error] CHART2Exception
[event under control of another op center and no override rights] ResourceControlConflict
[error] GUIException
[error] completed (failure reason)
[error] "Display Error"
[error]
completed (success)

Figure 150. GUITrafficEventModule:ExecuteResponse (Sequence Diagram)

3.12.2.14 GUITrafficEventModule:ExecuteResponseItem (Sequence Diagram)

This diagram shows how one or more response plan items are executed. The user clicks on the "Execute Selected Items" button or menu item, and the GUIResponsePlanItem's execute() method is called, which calls the ResponsePlanItem's execute(). If successful, the server will push a PlanStatusChanged event and the GUIResponsePlanItems will be updated to show the status of the individual items. Upon failure of individual items, a CommandStatusImpl object will be created to show the failure in the CommandFailures window.

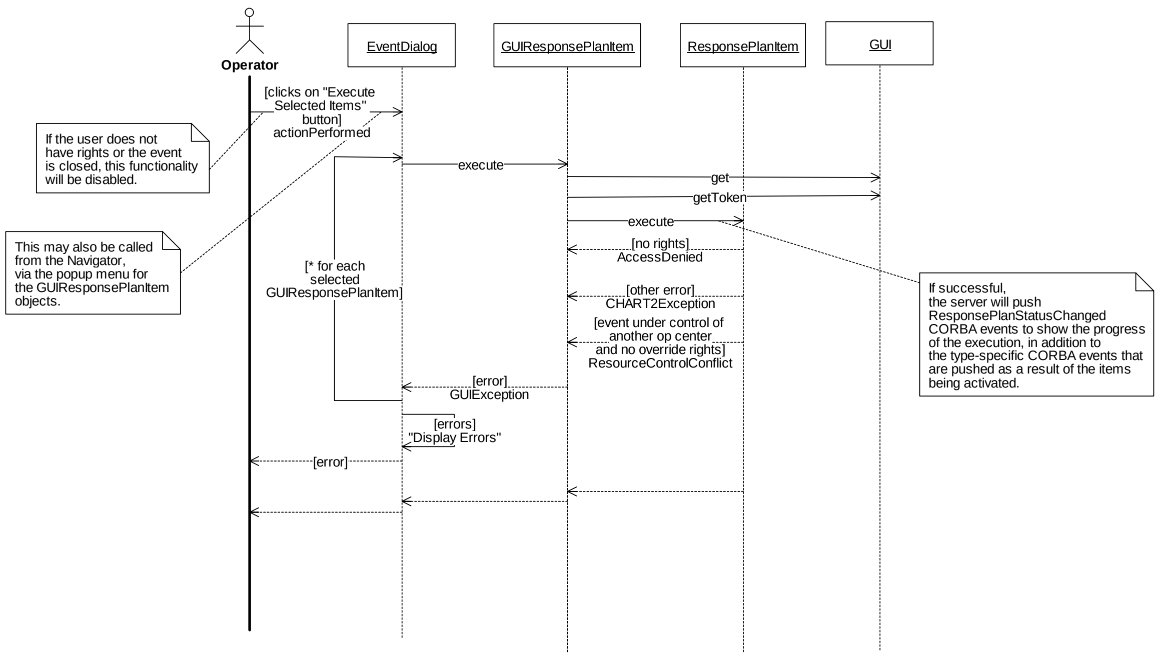

Operator
EventDialog
GUIResponsePlanItem
ResponsePlanItem
GUI
[clicks on "Execute Selected Items" button] actionPerformed
If the user does not have rights or the event is closed, this functionality will be disabled.
This may also be called from the Navigator, via the popup menu for the GUIResponsePlanItem objects.
execute
get
getToken
execute
[no rights] AccessDenied
[* for each selected GUIResponsePlanItem]
[other error] CHART2Exception
[event under control of another op center and no override rights] ResourceControlConflict
If successful, the server will push ResponsePlanStatusChanged CORBA events to show the progress of the execution, in addition to the type-specific CORBA events that are pushed as a result of the items being activated.
[error] GUIException
[errors] "Display Errors"
[error]

Figure 151. GUITrafficEventModule:ExecuteResponseItem (Sequence Diagram)

3.12.2.15 GUITrafficEventModule:GetEventHistoryText (Sequence Diagram)

This diagram shows how a log search is done. The user clicks on the search button in the EventDialog, and a search dialog is displayed and initialized with the settings from the last LogFilter (if any). The user enters the search criteria and presses the “OK” or “Search” button. This causes a LogFilter to be created and then an EventLogSearcher thread is created and started for the asynchronous search. The thread calls the getHistory() method of the GUITrafficEventHolder, which in turn calls the TrafficEvent. Results from the search (either errors or LogEntries) are stored in SearchErrorCommand objects or SearchEntriesFoundCommand objects and are invoked later on the main AWT event thread. (This is necessary to ensure proper interaction between the search dialog and the search results). If there are more entries, the LogIterator will be used to get them. When the search dialog receives a new batch of log entries, it will sort them to ensure that they are displayed in the correct order.

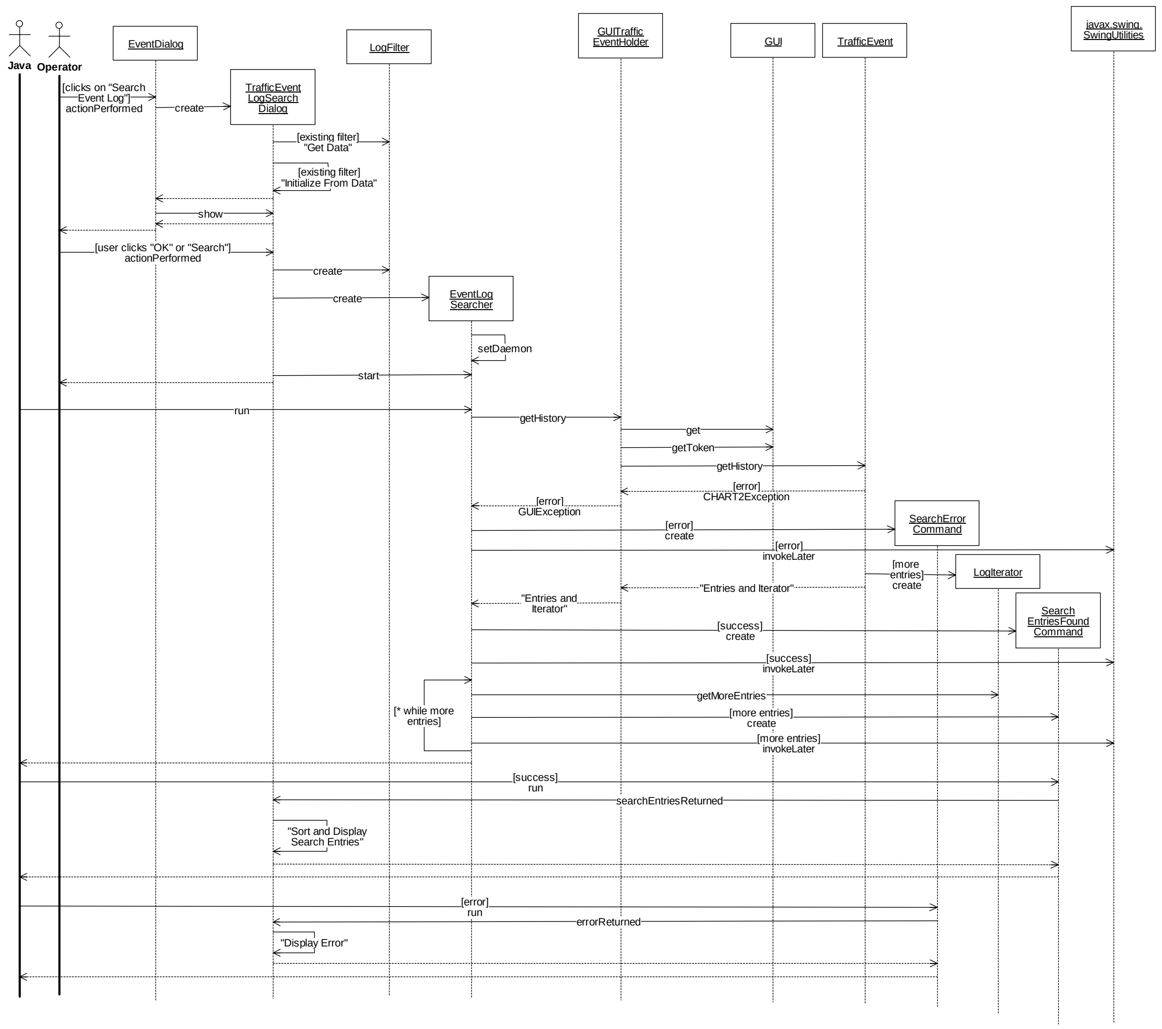


Figure 152. GUITrafficEventModule:GetEventHistoryText (Sequence Diagram)

[DCE:303]

3.12.2.16 GUITrafficEventModule:HandleEventCommLogEntryAdded (Sequence Diagram)

This diagram shows the processing that is done when a LogEntryAdded event is pushed to the GUI. The CommLogPushReceiver catches the event and calls the GUICommLog's entryAdded() method. The entry is added to the GUICommLog's cache and the GUICommLog calls the DataModel to notify all observers that the GUICommLog has been updated.

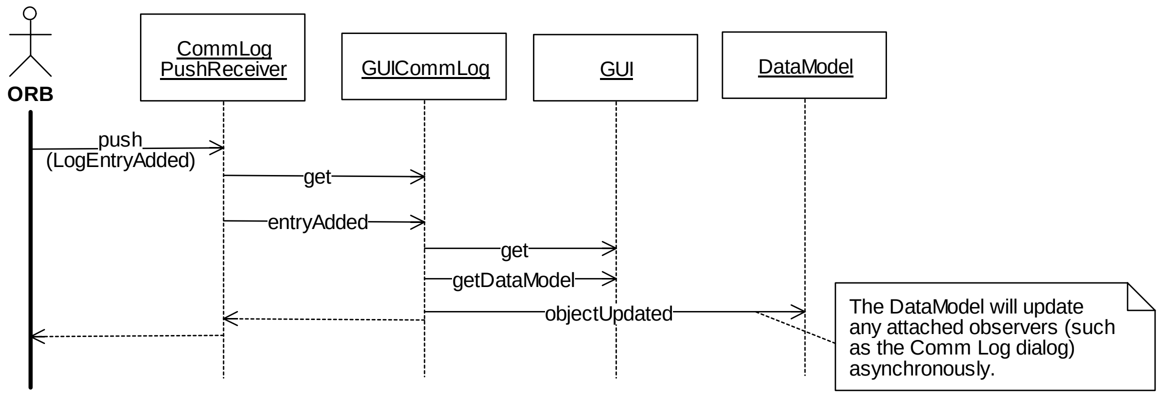


Figure 153. GUITrafficEventModule:HandleEventCommLogEntryAdded (Sequence Diagram)

3.12.2.17 GUITrafficEventModule:HandleEventEventAdded (Sequence Diagram)

This diagram shows the processing that occurs when a TrafficEventAdded CORBA event is pushed to the GUI. The TrafficEventPushReceiver receives the event, and creates a GUITrafficEventHolder object if one does not already exist in the DataModel with the same ID. (The GUITrafficEventHolder may already exist if this GUI just created a new one before calling the TrafficEvent). The GUITrafficEventHolder will create a GUITrafficEvent, whose type depends on the type of BasicEventData passed in. The GUITrafficEventHolder and GUITrafficEvent are added to the DataModel. Any log entries are then added to the GUITrafficEventHolder object. As it is a new event, there may or may not be existing log entries; however, there will not be any response plan items or response participations.

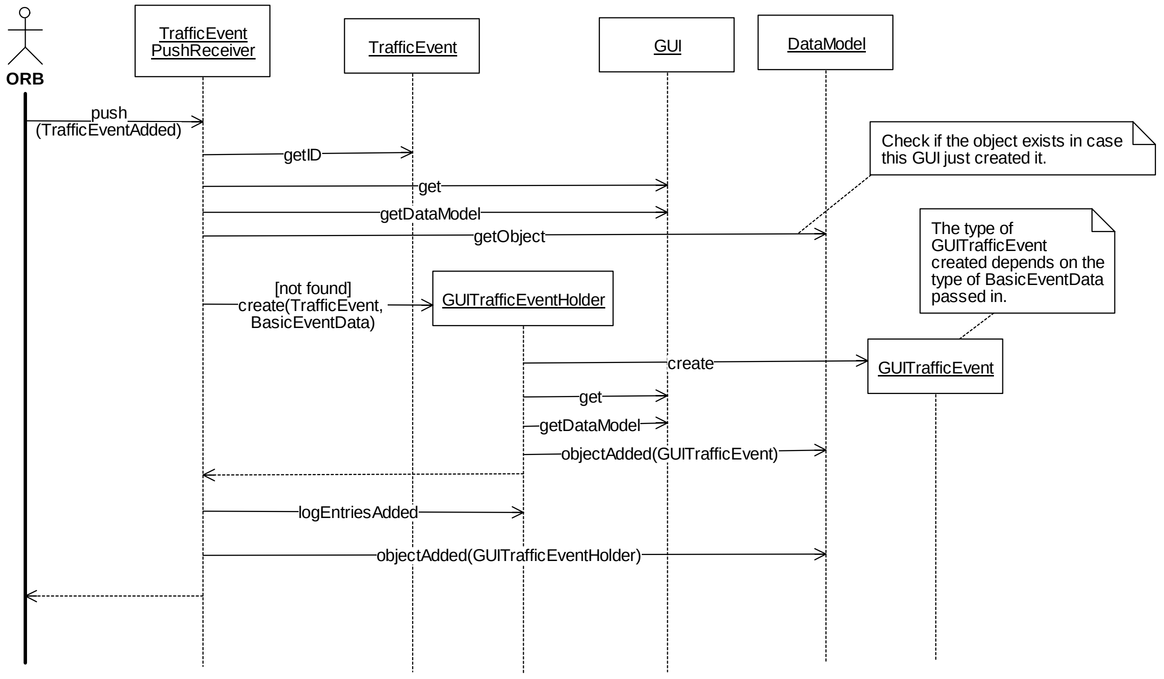

ORB
TrafficEvent PushReceiver
TrafficEvent
GUI
DataModel
push (TrafficEventAdded)
getID
get
getDataModel
getObject
Check if the object exists in case this GUI just created it.
[not found] create(TrafficEvent, BasicEventData)
GUITrafficEventHolder
The type of GUITrafficEvent created depends on the type of BasicEventData passed in.
create
GUITrafficEvent
get
getDataModel
objectAdded(GUITrafficEvent)
logEntriesAdded
objectAdded(GUITrafficEventHolder)

Figure 154. GUITrafficEventModule:HandleEventEventAdded (Sequence Diagram)

3.12.2.18 GUITrafficEventModule:HandleEventEventAssociated (Sequence Diagram)

This diagram shows the processing that occurs when a TrafficEventAssociated or TrafficEventAssociationRemoved event is received in the GUI. The TrafficEventPushReceiver receives the event, then it calls the GUITrafficEventModule to find the association. If the association was added and the TrafficEventAssociation object did not exist, a new TrafficEventAssociation is created and added to the DataModel. If the TrafficEventAssociation did exist and the association was removed, then it will be removed from the DataModel.

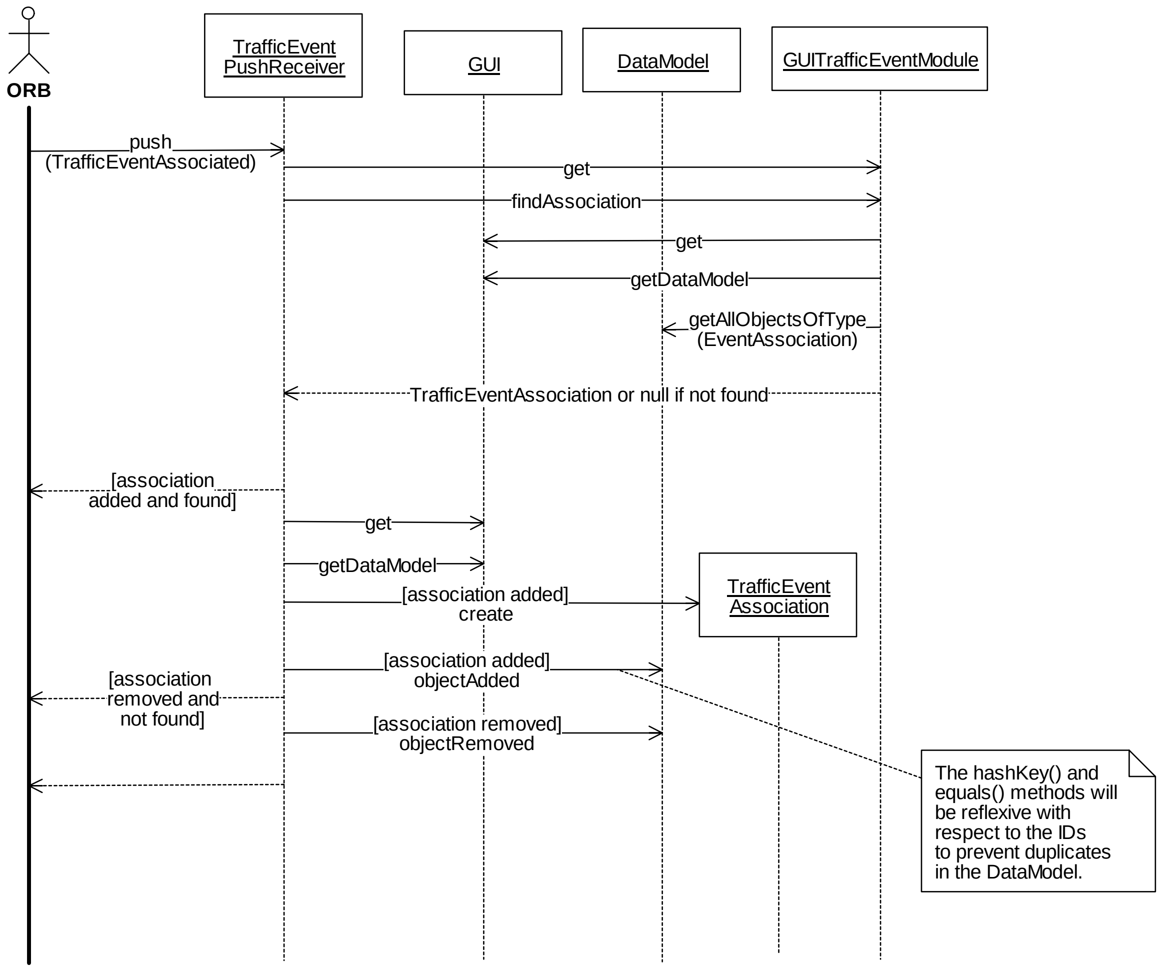

ORB
TrafficEvent PushReceiver
GUI
DataModel
GUITrafficEventModule
push (TrafficEventAssociated)
get
findAssociation
get
getDataModel
getAllObjectsOfType (EventAssociation)
TrafficEventAssociation or null if not found
[association added and found]
get
getDataModel
[association added] create
TrafficEvent Association
[association added] objectAdded
[association removed and not found]
[association removed] objectRemoved
The hashKey() and equals() methods will be reflexive with respect to the IDs to prevent duplicates in the DataModel.

Figure 155. GUITrafficEventModule:HandleEventEventAssociated (Sequence Diagram)

3.12.2.19 GUITrafficEventModule:HandleEventEventClosed (Sequence Diagram)

This diagram shows the processing that occurs if the event is closed and a TrafficEventClosed event is received in the GUI. The TrafficEventPushReceiver receives the pushed event, and calls the GUITrafficEventHolder to inform it that it's closed. The GUITrafficEventHolder updates any state data and then calls the DataModel's objectUpdated() method to notify all interested observers of the change.

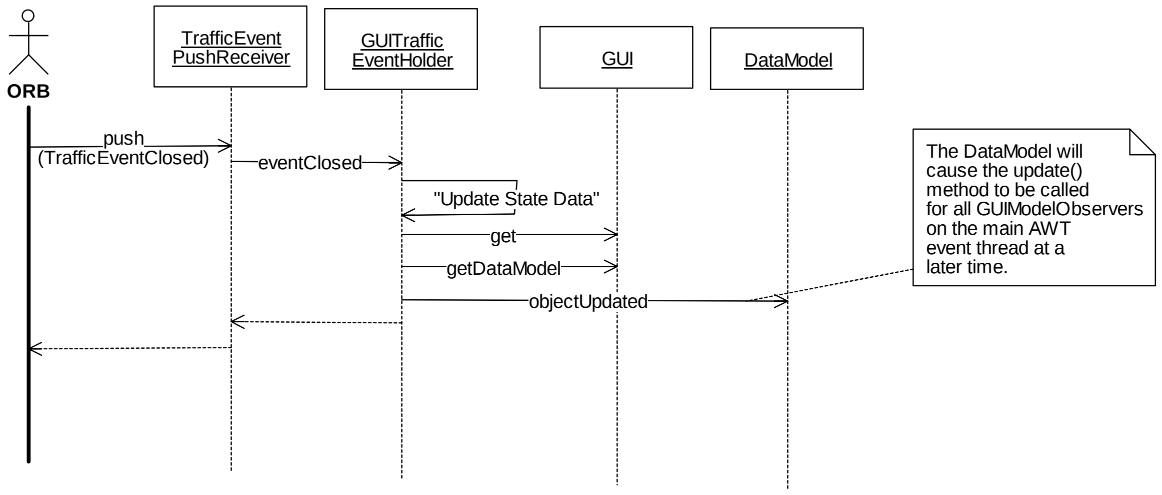

ORB
TrafficEvent PushReceiver
GUITraffic EventHolder
GUI
DataModel
push (TrafficEventClosed)
eventClosed
"Update State Data"
get
getDataModel
objectUpdated
The DataModel will cause the update() method to be called for all GUIModelObservers on the main AWT event thread at a later time.

Figure 156. GUITrafficEventModule:HandleEventEventClosed (Sequence Diagram)

3.12.2.20 GUITrafficEventModule:HandleEventEventDeleted (Sequence Diagram)

This diagram shows the processing that happens when a traffic event is removed from the system and a TrafficEventDeleted CORBA event is pushed to the GUI. The TrafficEventPushReceiver catches the event and calls the GUITrafficEventHolder to inform it that the event was removed. The GUITrafficEventHolder calls the GUITrafficEvent to clean itself up. This will cause all panels in the TrafficEventDialog which apply to the event to be cleaned up and the references to the event removed. Any references held by the GUITrafficEventHolder will be cleaned up, and the GUITrafficEvent and GUITrafficEventHolder will be removed from the DataModel. All input on the EventDialog will be disabled. A message will be displayed to the user explaining that the event was removed from the system and the input will be disabled in the event history log.

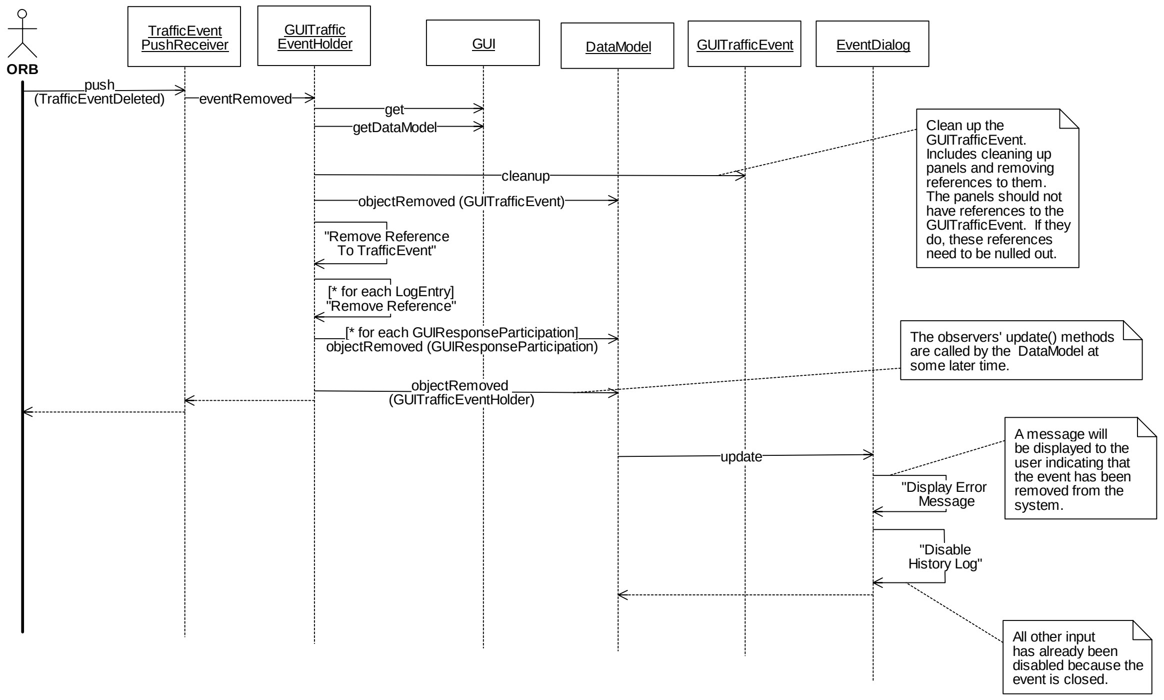

ORB
TrafficEvent PushReceiver
GUITraffic EventHolder
GUI
DataModel
GUITrafficEvent
EventDialog
push (TrafficEventDeleted)
eventRemoved
get
getDataModel
cleanup
objectRemoved (GUITrafficEvent)
"Remove Reference To TrafficEvent"
[* for each LogEntry] "Remove Reference"
[* for each GUIResponseParticipation] objectRemoved (GUIResponseParticipation)
objectRemoved (GUITrafficEventHolder)
update
"Display Error Message
"Disable History Log"
Clean up the GUITrafficEvent. Includes cleaning up panels and removing references to them. The panels should not have references to the GUITrafficEvent. If they do, these references need to be nulled out.
The observers' update() methods are called by the DataModel at some later time.
A message will be displayed to the user indicating that the event has been removed from the system.
All other input has already been disabled because the event is closed.

Figure 157. GUITrafficEventModule:HandleEventEventDeleted (Sequence Diagram)

3.12.2.21 GUITrafficEventModule:HandleEventEventTypeChanged (Sequence Diagram)

This diagram shows the processing that occurs when a TrafficEventTypeChanged event is received on the TrafficEvent channel. The TrafficEventPushReceiver receives the event, and notifies the GUITrafficEventHolder that the type has changed. The GUITrafficEventHolder creates a new GUITrafficEvent and replaces its reference to the event. The old GUITrafficEvent is removed from the DataModel and the new one is added, and the GUITrafficEventHolder calls the DataModel to inform interested observers that it has changed. The EventDialog will receive an update from the DataModel and will remove all of the panels for the old GUITrafficEvent, and the old GUITrafficEvent will clean itself up and be deleted. The new GUITrafficEvent will supply all of the panels for the EventDialog, and these panels will be added to the dialog and displayed.

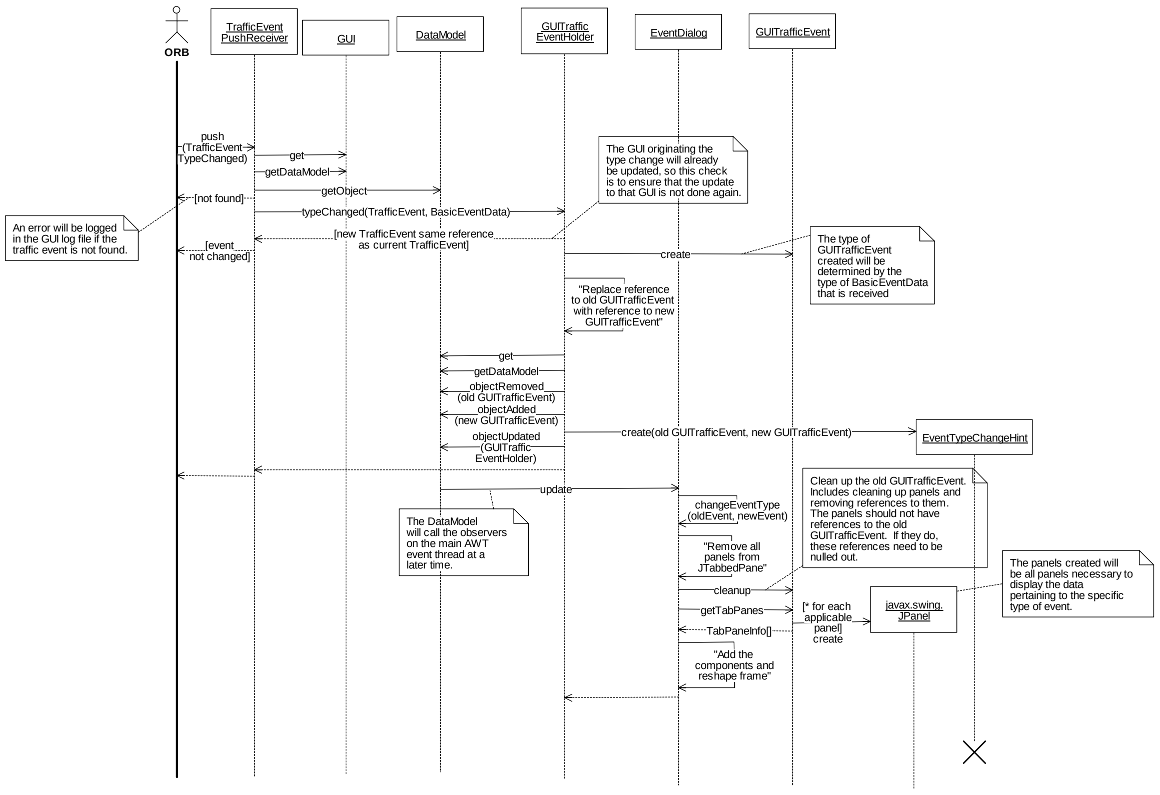
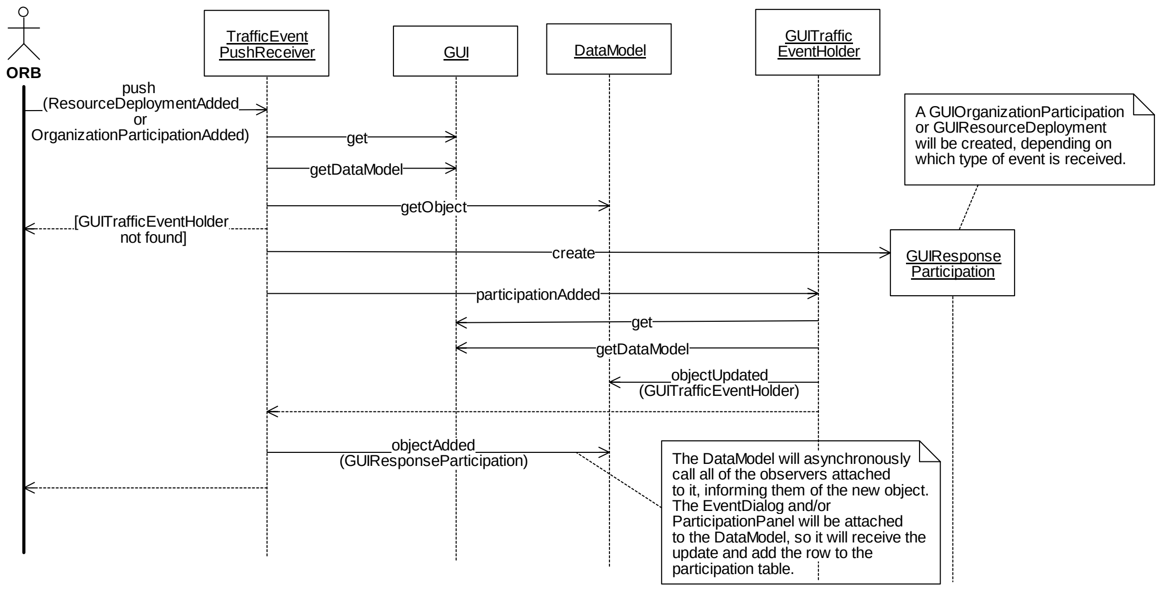

ORB
TrafficEvent PushReceiver
GUI
DataModel
GUITraffic EventHolder
EventDialog
GUITrafficEvent
push (TrafficEvent TypeChanged)
get
getDataModel
getObject
[not found]
typeChanged(TrafficEvent, BasicEventData)
The GUI originating the type change will already be updated, so this check is to ensure that the update to that GUI is not done again.
An error will be logged in the GUI log file if the traffic event is not found.
[event not changed]
[new TrafficEvent same reference as current TrafficEvent]
create
The type of GUITrafficEvent created will be determined by the type of BasicEventData that is received
"Replace reference to old GUITrafficEvent with reference to new GUITrafficEvent"
get
getDataModel
objectRemoved (old GUITrafficEvent)
objectAdded (new GUITrafficEvent)
create(old GUITrafficEvent, new GUITrafficEvent)
EventTypeChangeHint
objectUpdated (GUITraffic EventHolder)
update
changeEventType (oldEvent, newEvent)
Clean up the old GUITrafficEvent. Includes cleaning up panels and removing references to them. The panels should not have references to the old GUITrafficEvent. If they do, these references need to be nulled out.
The DataModel will call the observers on the main AWT event thread at a later time.
"Remove all panels from JTabbedPane"
cleanup
The panels created will be all panels necessary to display the data pertaining to the specific type of event.
getTabPanes
[* for each applicable panel] create
javax.swing. JPanel
TabPaneInfo[]
"Add the components and reshape frame"

Figure 158. GUITrafficEventModule:HandleEventEventTypeChanged (Sequence Diagram)

3.12.2.22 GUITrafficEventModule:HandleEventResponseParticipationAdded (Sequence Diagram)

This diagram shows the processing that occurs when a ResourceDeploymentAdded or OrganizationParticipationAdded event is received on the TrafficEvent event channel. The TrafficEventPushReceiver receives the event and creates a GUIOrganizationParticipation or GUIResourceDeployment, and adds it to the GUITrafficEventHolder. The GUITrafficEventHolder calls the DataModel to inform all interested observers that it has changed, so that the EventDialog can be updated with the new participation. The new object will also be added to the DataModel.



ORB
TrafficEvent PushReceiver
GUI
DataModel
GUITraffic EventHolder
push (ResourceDeploymentAdded or OrganizationParticipationAdded)
get
getDataModel
getObject
[GUITrafficEventHolder not found]
create
GUIResponse Participation
A GUIOrganizationParticipation or GUIResourceDeployment will be created, depending on which type of event is received.
participationAdded
get
getDataModel
objectUpdated (GUITrafficEventHolder)
objectAdded (GUIResponseParticipation)
The DataModel will asynchronously call all of the observers attached to it, informing them of the new object. The EventDialog and/or ParticipationPanel will be attached to the DataModel, so it will receive the update and add the row to the participation table.

Figure 159. GUITrafficEventModule:HandleEventResponseParticipationAdded (Sequence Diagram)

3.12.2.23 GUITrafficEventModule:HandleEventResponseParticipationRemoved (Sequence Diagram)

This diagram shows the processing that occurs when a ParticipationRemoved event is received by the TrafficEvent channel. The TrafficEventPushReceiver receives the event, gets the GUIResponseParticipation object from the DataModel, and asks it for the GUITrafficEventHolder. The GUITrafficEventHolder removes the GUIResponseParticipation from itself and calls the DataModel to update any observers (notably the EventDialog). The GUIResponseParticipation object is also removed from the DataModel.

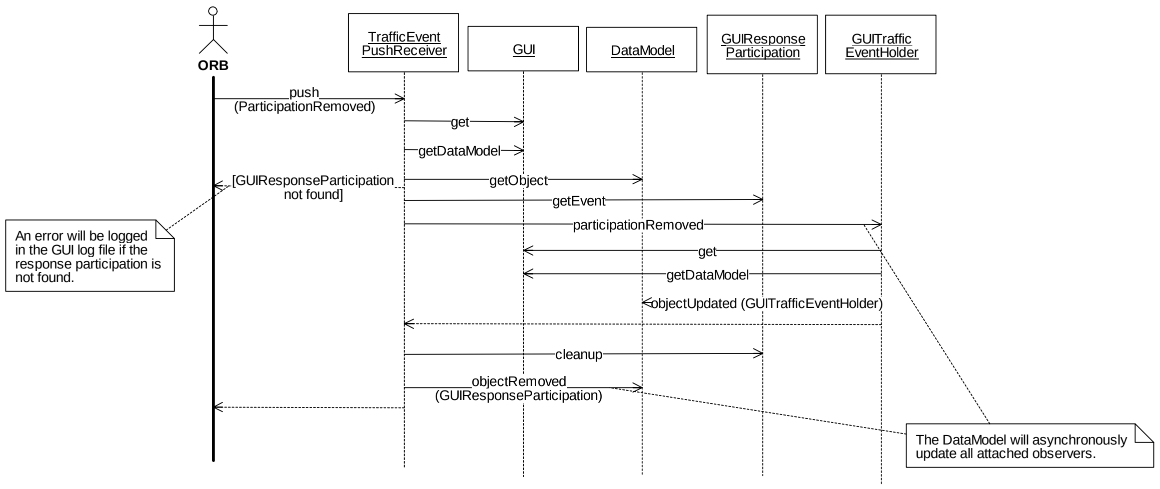
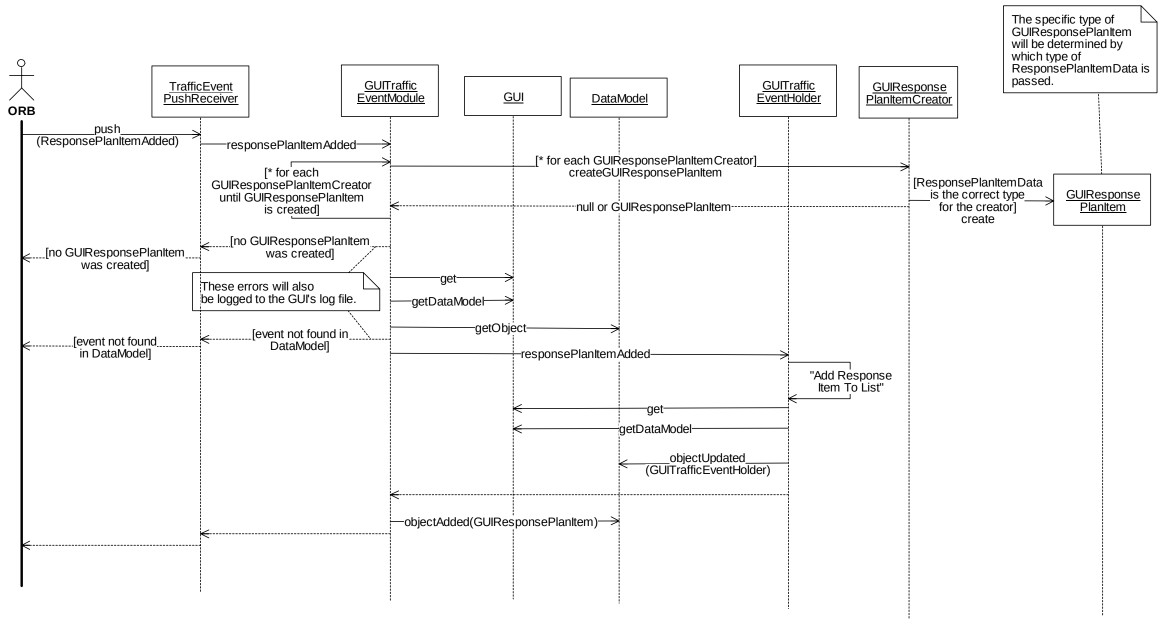

ORB
TrafficEvent PushReceiver
GUI
DataModel
GUIResponse Participation
GUITraffic EventHolder
push (ParticipationRemoved)
get
getDataModel
getObject
[GUIResponseParticipation not found]
getEvent
participationRemoved
get
getDataModel
objectUpdated (GUITrafficEventHolder)
cleanup
objectRemoved (GUIResponseParticipation)
An error will be logged in the GUI log file if the response participation is not found.
The DataModel will asynchronously update all attached observers.

Figure 160. GUITrafficEventModule:HandleEventResponseParticipationRemoved (Sequence Diagram)

3.12.2.24 GUITrafficEventModule:HandleEventResponsePlanItemAdded (Sequence Diagram)

This diagram shows the processing that occurs when a ResponsePlanItemAdded event is received on the TrafficEvent event channel. The TrafficEventPushReceiver receives the event and calls the GUITrafficEventModule to tell it that a new ResponsePlanItem has been added. The module then calls each of the installed GUIResponsePlanItemCreators to create a GUIResponsePlanItem wrapper for the ResponsePlanItem. (A GUIResponsePlanItemCreator will examine the type of the ResponsePlanItemData and if it recognizes the type, it will create the wrapper). The module then gets the GUITrafficEventHolder from the DataModel and adds the GUIResponsePlanItem to its list, and updates itself through the DataModel to inform any interested observers of the change. The GUIResponsePlanItem is also added to the DataModel.

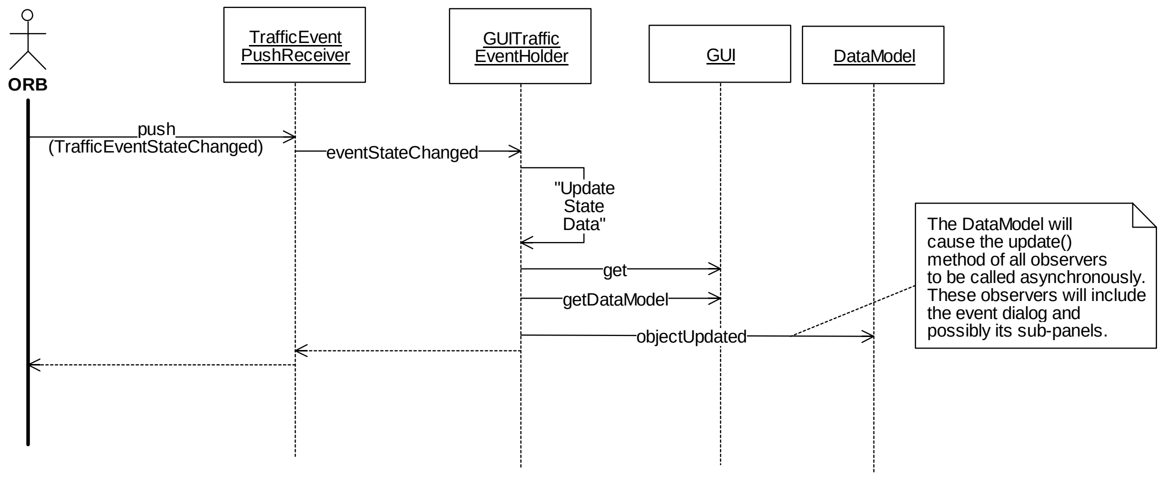


ORB
TrafficEvent PushReceiver
GUITraffic EventModule
GUI
DataModel
GUITraffic EventHolder
GUIResponse PlanItemCreator
The specific type of GUIResponsePlanItem will be determined by which type of ResponsePlanItemData is passed.
push (ResponsePlanItemAdded)
responsePlanItemAdded
[* for each GUIResponsePlanItemCreator] createGUIResponsePlanItem
[* for each GUIResponsePlanItemCreator until GUIResponsePlanItem is created]
[ResponsePlanItemData is the correct type for the creator] create
GUIResponse PlanItem
null or GUIResponsePlanItem
[no GUIResponsePlanItem was created]
[no GUIResponsePlanItem was created]
These errors will also be logged to the GUI's log file.
get
getDataModel
getObject
[event not found in DataModel]
[event not found in DataModel]
responsePlanItemAdded
"Add Response Item To List"
get
getDataModel
objectUpdated (GUITrafficEventHolder)
objectAdded(GUIResponsePlanItem)

Figure 161. GUITrafficEventModule:HandleEventResponsePlanItemAdded (Sequence Diagram)

3.12.2.25 GUITrafficEventModule:HandleEventTrafficEventStateChanged (Sequence Diagram)

This diagram shows the processing that occurs if the event data is changed and a TrafficEventStateChanged event is received in the GUI. The TrafficEventPushReceiver receives the pushed event, and calls the GUITrafficEventHolder to inform it that the state has changed. The GUITrafficEventHolder updates any state data and then calls the DataModel's objectUpdated() method to notify all interested observers of the change.



ORB
TrafficEvent PushReceiver
GUITraffic EventHolder
GUI
DataModel
push (TrafficEventStateChanged)
eventStateChanged
"Update State Data"
get
getDataModel
objectUpdated
The DataModel will cause the update() method of all observers to be called asynchronously. These observers will include the event dialog and possibly its sub-panels.

Figure 162. GUITrafficEventModule:HandleEventTrafficEventStateChanged (Sequence Diagram)

3.12.2.26 GUITrafficEventModule:Login (Sequence Diagram)

This diagram shows the processing that is done when the user logs in. The GUI calls the loggedIn() method of the GUITrafficEventModule. The module gets the toolbar and enables the “Comm Log” button if the user has rights.

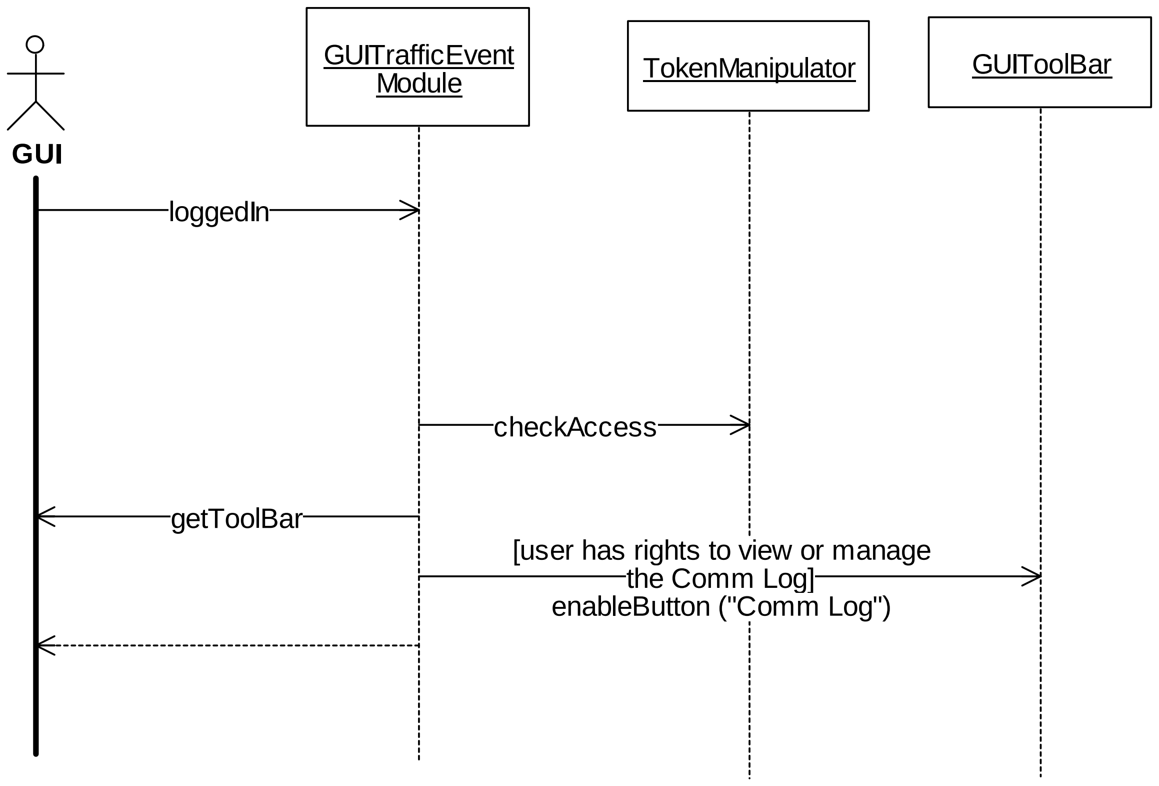


Figure 163. GUITrafficEventModule:Login (Sequence Diagram)

3.12.2.27 GUITrafficEventModule:Logout (Sequence Diagram)

This diagram shows the processing that is performed at logout. The GUI calls the GUITrafficEvent's loggedOut() method. Currently no other work is done at logout.

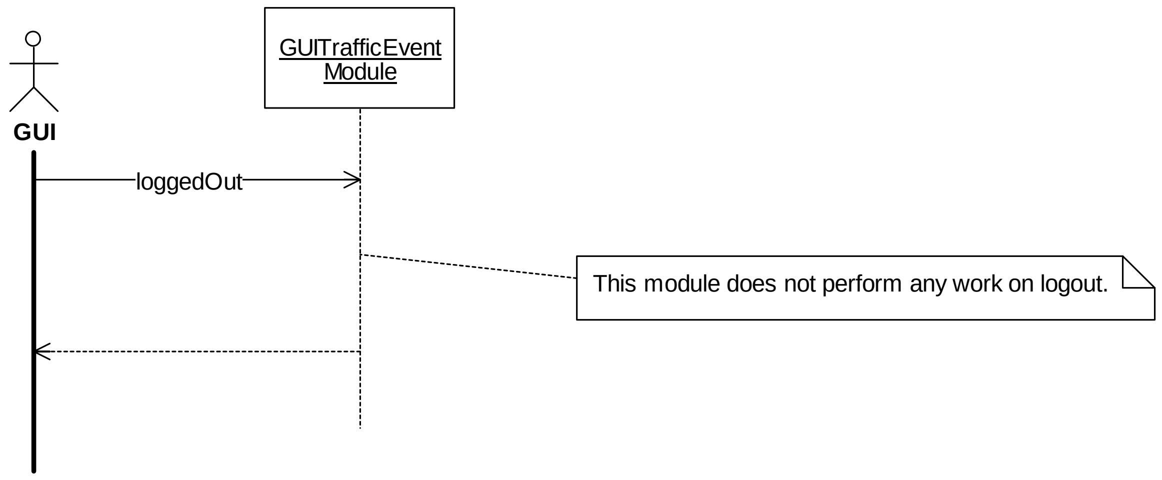


Figure 164. GUITrafficEventModule:Logout (Sequence Diagram)

3.12.2.28 GUITrafficEventModule:ModifyResponseParticipationData (Sequence Diagram)

This diagram shows an example of how a ResponseParticipation will be modified from the GUI. The user may click on the “Notified” check box, which will cause the setNotified() method of the GUIResponseParticipation object to be called, which will in turn call the corresponding setNotified() method of the ResponseParticipation CORBA interface. If successful, an OrganizationParticipationChanged or ResourceDeploymentChanged event will be pushed by the server.

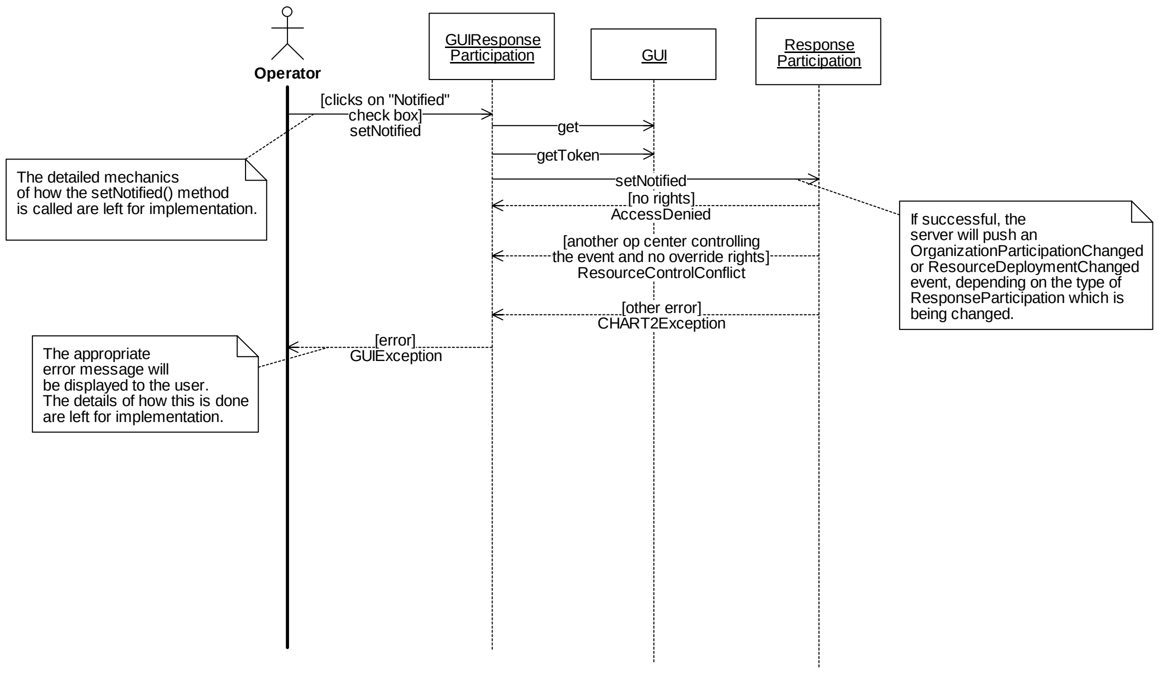

Operator
GUIResponse Participation
GUI
Response Participation
[clicks on "Notified" check box] setNotified
get
getToken
setNotified
[no rights] AccessDenied
[another op center controlling the event and no override rights] ResourceControlConflict
[other error] CHART2Exception
[error] GUIException
The detailed mechanics of how the setNotified() method is called are left for implementation.
If successful, the server will push an OrganizationParticipationChanged or ResourceDeploymentChanged event, depending on the type of ResponseParticipation which is being changed.
The appropriate error message will be displayed to the user. The details of how this is done are left for implementation.

Figure 165. GUITrafficEventModule:ModifyResponseParticipationData (Sequence Diagram)

3.12.2.29 GUITrafficEventModule:ModifyResponsePlanItemMessage (Sequence Diagram)

This diagram shows how a message might be modified for a ResponsePlanItem. (NOTE - this operation depends on the type of ResponsePlanItem, so the design may differ somewhat for different types). The operator clicks on the “Edit Message” menu item on the GUIResponsePlanItem’s context menu. The GUIResponsePlanItem catches the command and calls doProperties() on itself. This creates a message editor, which calls the GUIResponsePlanItem to get the item data, which in turn calls the ResponsePlanItem CORBA interface. There may be no data returned, in the case that the message has not been set. In this case, a new ResponsePlanItemData will be created by the message editor. The message editor will then initialize itself with the ResponsePlanItemData (either retrieved from the ResponsePlanItem or just created). When the user modifies the message and clicks “OK” in the message editor dialog, the dialog sets the data in a ResponsePlanItemData and calls the GUIResponsePlanItem to set the item data, which then calls the ResponsePlanItem. If successful, the message editor will be closed; otherwise, it will remain open and an error will be displayed. A ResponsePlanItemModified event will also be pushed to all of the GUIs if it is successful.

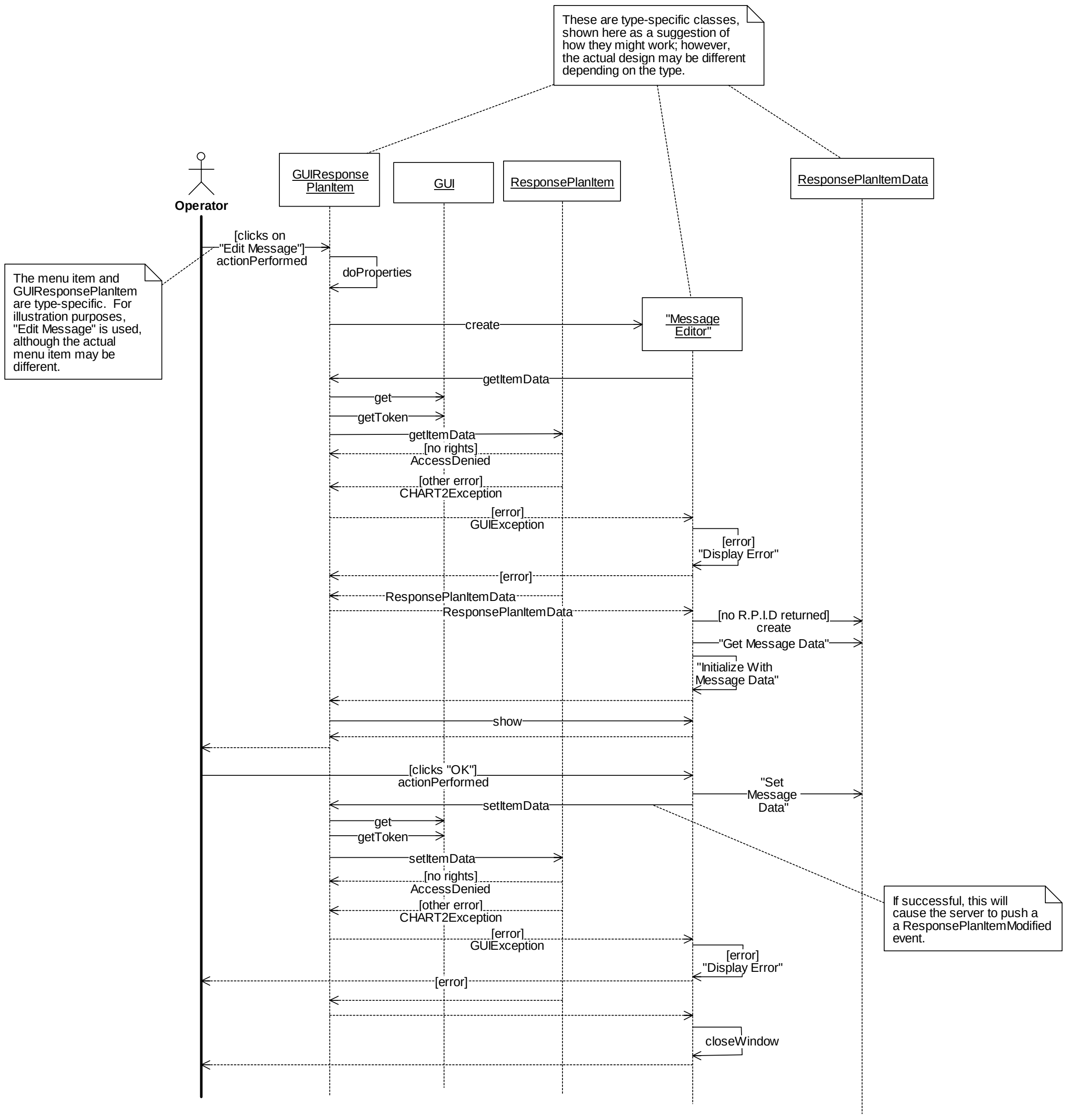

These are type-specific classes, shown here as a suggestion of how they might work; however, the actual design may be different depending on the type.
The menu item and GUIResponsePlanItem are type-specific. For illustration purposes, "Edit Message" is used, although the actual menu item may be different.
If successful, this will cause the server to push a a ResponsePlanItemModified event.
Operator
GUIResponse PlanItem
GUI
ResponsePlanItem
ResponsePlanItemData
"Message Editor"
[clicks on "Edit Message"] actionPerformed
doProperties
create
getItemData
get
getToken
getItemData
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
ResponsePlanItemData
ResponsePlanItemData
[no R.P.I.D returned] create
"Get Message Data"
"Initialize With Message Data"
show
[clicks "OK"] actionPerformed
"Set Message Data"
setItemData
get
getToken
setItemData
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] "Display Error"
[error]
closeWindow

Figure 166. GUITrafficEventModule:ModifyResponsePlanItemMessage (Sequence Diagram)

3.12.2.30 GUITrafficEventModule:RemoveItemFromResponse (Sequence Diagram)

This diagram shows how a response plan item is removed from a response plan. The user clicks on the “Remove” button in the EventDialog, or from the GUIResponsePlanItem’s context menu. The GUIResponsePlanItem’s remove() method is called, which calls the ResponsePlanItem’s remove() method. The server will push a ResponsePlanItemsRemoved event if it is successful; otherwise, an error message will be displayed in the EventDialog or in the Command Failures window if the command was not invoked from the dialog.

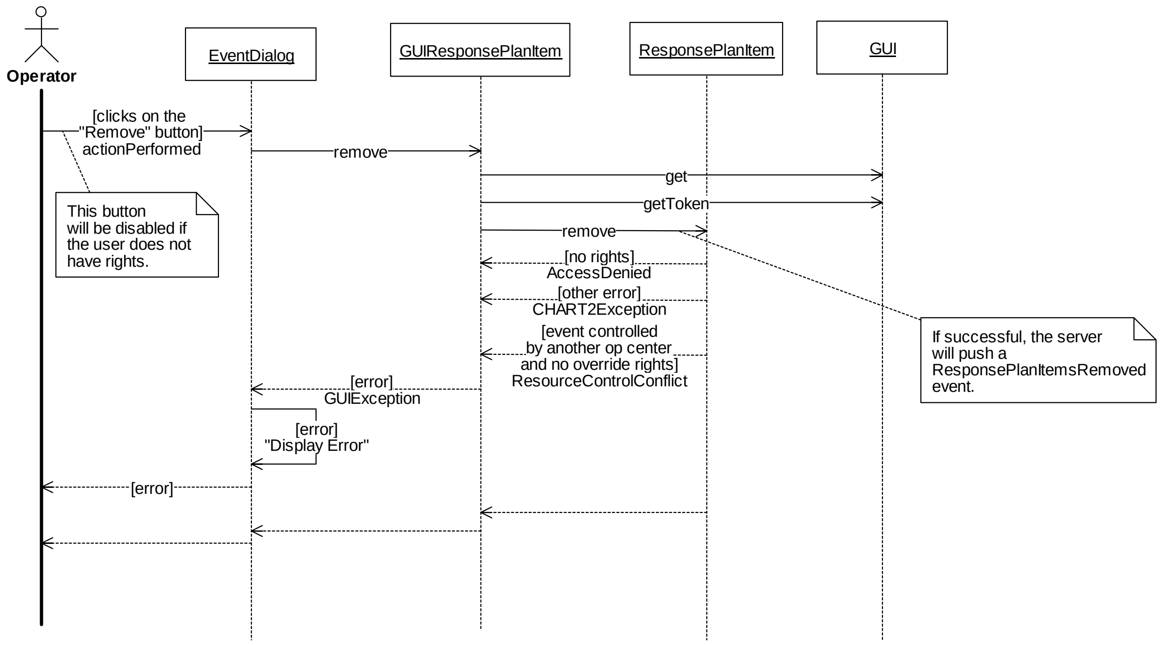
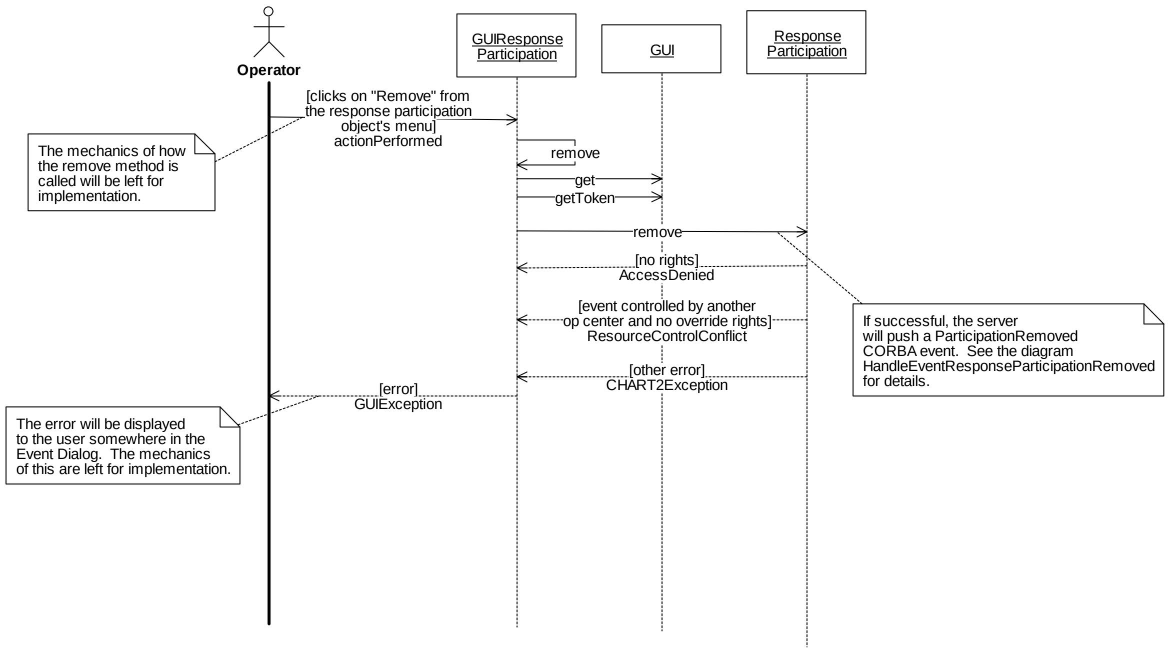

Operator
EventDialog
GUIResponsePlanItem
ResponsePlanItem
GUI
[clicks on the "Remove" button] actionPerformed
remove
get
getToken
remove
This button will be disabled if the user does not have rights.
[no rights] AccessDenied
[other error] CHART2Exception
[event controlled by another op center and no override rights] ResourceControlConflict
If successful, the server will push a ResponsePlanItemsRemoved event.
[error] GUIException
[error] "Display Error"
[error]

Figure 167. GUITrafficEventModule:RemoveItemFromResponse (Sequence Diagram)

3.12.2.31 GUITrafficEventModule:RemoveResponseParticipation (Sequence Diagram)

This diagram shows how a ResponseParticipation is removed. The user clicks on the "Remove" menu item on the GUIResponseParticipation's context menu, and it calls remove() on itself. This calls the ResponseParticipation CORBA interface remove() method. This will cause the server to push a ParticipationRemoved CORBA event if successful; otherwise, an error is displayed in the EventDialog. See the sequence diagram: HandleEventResponseParticipationRemoved for details on the handling of the event.



Operator
GUIResponse Participation
GUI
Response Participation
[clicks on "Remove" from the response participation object's menu] actionPerformed
remove
get
getToken
remove
[no rights] AccessDenied
[event controlled by another op center and no override rights] ResourceControlConflict
[other error] CHART2Exception
[error] GUIException
The mechanics of how the remove method is called will be left for implementation.
The error will be displayed to the user somewhere in the Event Dialog. The mechanics of this are left for implementation.
If successful, the server will push a ParticipationRemoved CORBA event. See the diagram HandleEventResponseParticipationRemoved for details.

Figure 168. GUITrafficEventModule:RemoveResponseParticipation (Sequence Diagram)

3.12.2.32 GUITrafficEventModule:SearchCommLog (Sequence Diagram)

This diagram shows how a log search is done. The user clicks on the search button in the CommLogDialog, and a search dialog is displayed and initialized with the settings from the last LogFilter (if any). The user enters the search criteria and presses the “OK” or “Search” button. This causes a LogFilter to be created and then a CommLogSearcher thread is created and started for the asynchronous search. The thread calls the getEntries() method of the GUICommLog, which in turn calls the CommLog object(s) it contains. Results from the search (either errors or LogEntries) are stored in SearchErrorCommand objects or SearchEntriesFoundCommand objects and are invoked later on the main AWT event thread. (This is necessary to ensure proper interaction between the search dialog and the search results). If there are more entries, the LogIterator will be used to get them. When the search dialog receives a new batch of log entries, it will sort them to ensure that they are displayed in the correct order.

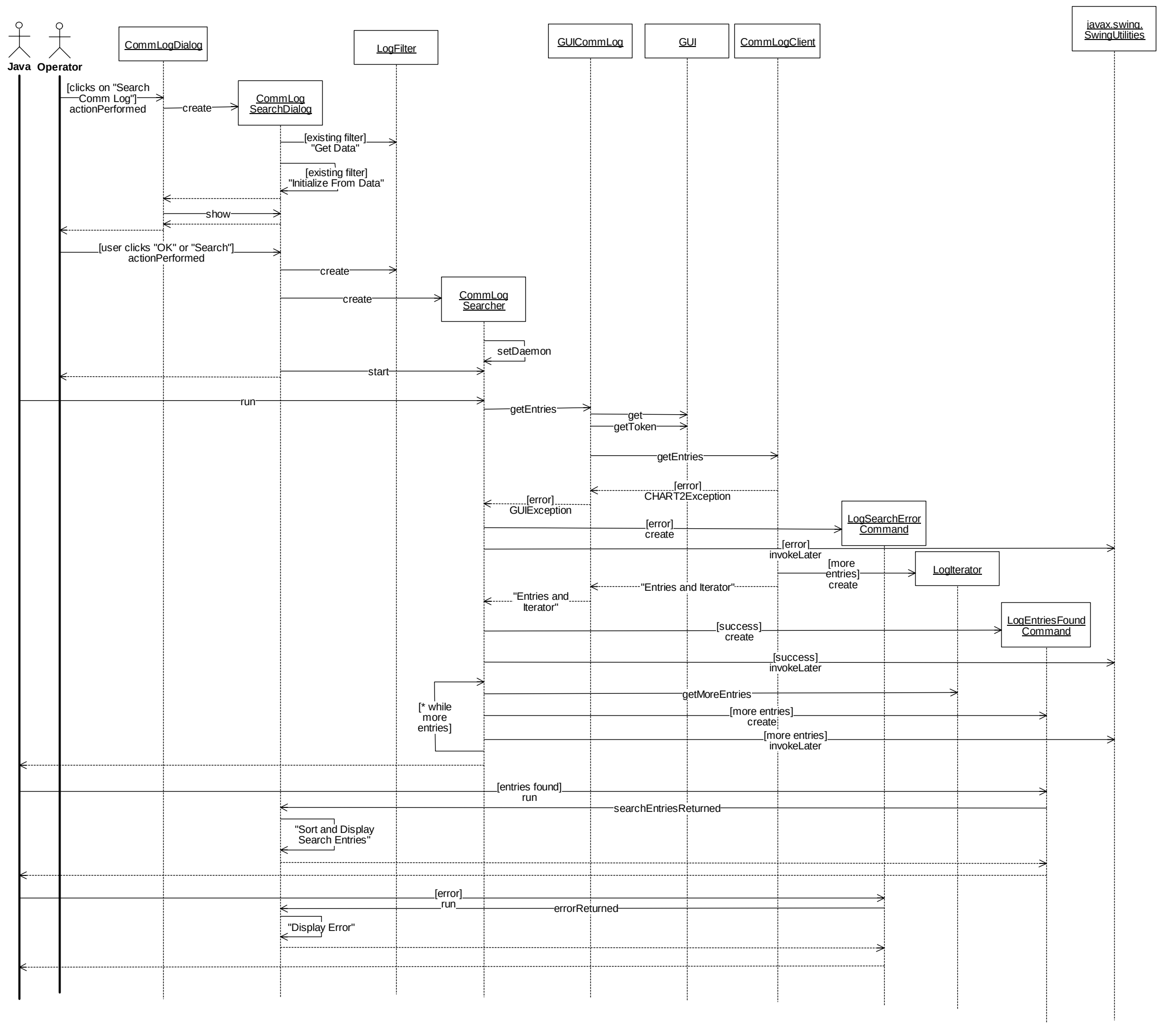


Figure 169. GUITrafficEventModule:SearchCommLog (Sequence Diagram)

[DCE:337]

3.12.2.33 GUITrafficEventModule:SetLaneConfiguration (Sequence Diagram)

This diagram shows how a lane configuration is set. During discovery, the possible standard lane configurations were obtained from the TrafficEventFactory objects and stored in the GUITrafficEventModule. The operator will be able to select one of the lane configurations. Then the operator clicks on “Set Lane Configuration”, which gets the lane configuration data for each lane in the panel. A LaneConfiguration object is created, and the GUITrafficEvent’s setLaneConfiguration() method is called. This calls the RoadwayEvent interface to set the lane configuration. If successful, the server will push a LaneConfigurationChanged event.

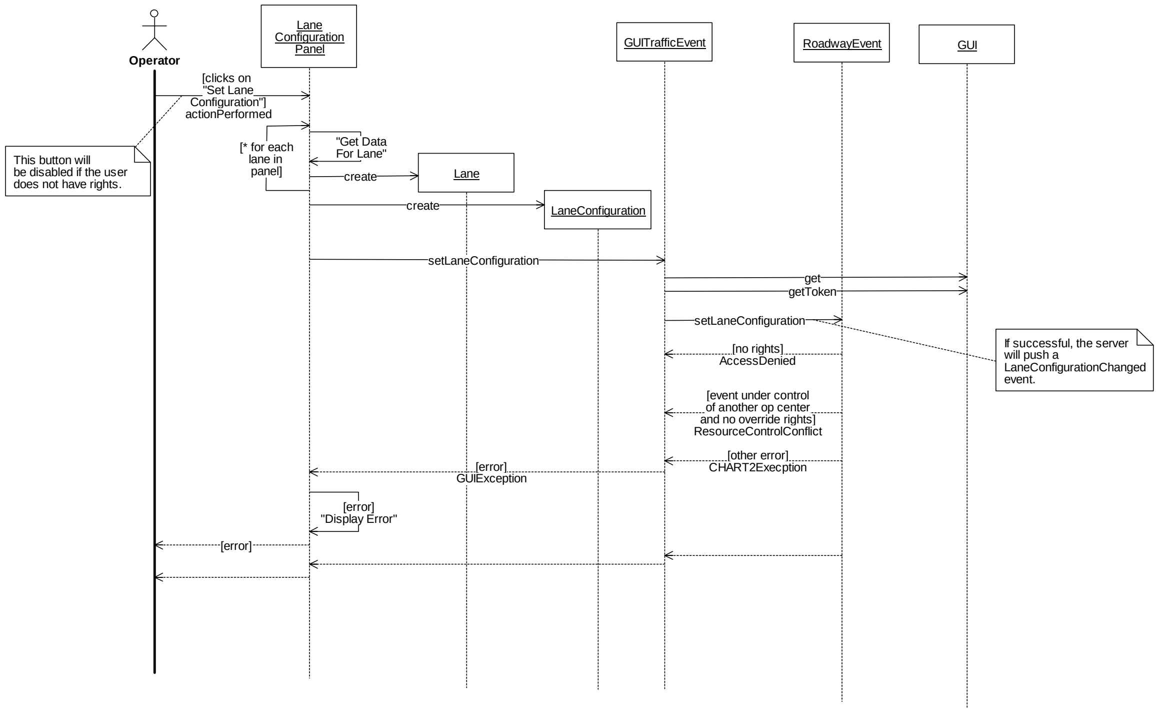

Operator
Lane Configuration Panel
GUITrafficEvent
RoadwayEvent
GUI
[clicks on "Set Lane Configuration"] actionPerformed
This button will be disabled if the user does not have rights.
[* for each lane in panel]
"Get Data For Lane"
create
Lane
create
LaneConfiguration
setLaneConfiguration
get
getToken
setLaneConfiguration
If successful, the server will push a LaneConfigurationChanged event.
[no rights] AccessDenied
[event under control of another op center and no override rights] ResourceControlConflict
[other error] CHART2Execption
[error] GUIException
[error] "Display Error"
[error]

Figure 170. GUITrafficEventModule:SetLaneConfiguration (Sequence Diagram)

3.12.2.34 GUITrafficEventModule:Shutdown (Sequence Diagram)

This diagram shows the processing that happens at shutdown. The GUI calls the GUITrafficEventModule's shutdown() method, which deactivates the TrafficEventPushReceiver and CommLogPushReceiver objects.

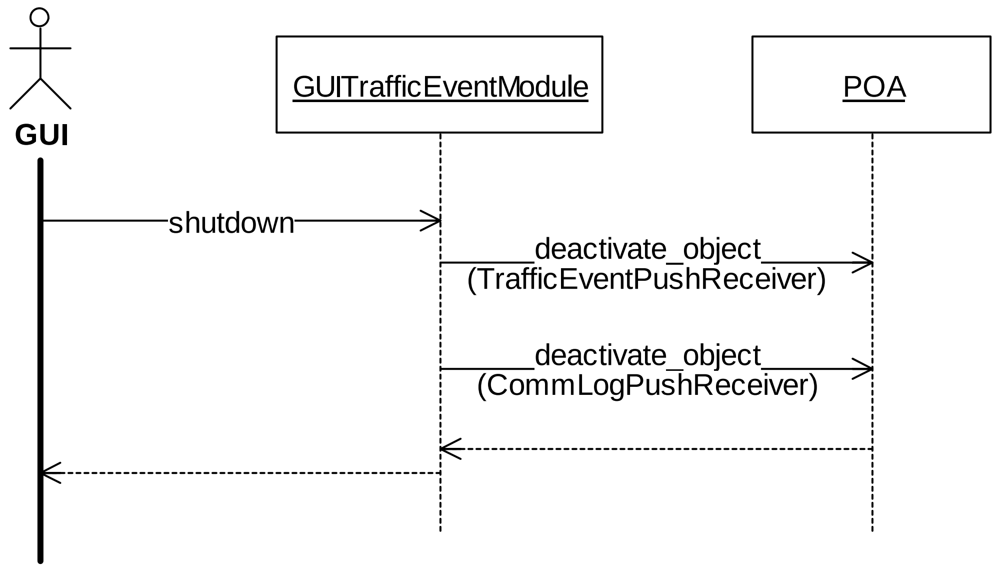

GUI
GUITrafficEventModule
POA
shutdown
deactivate_object (TrafficEventPushReceiver)
deactivate_object (CommLogPushReceiver)

Figure 171. GUITrafficEventModule:Shutdown (Sequence Diagram)

3.12.2.35 GUITrafficEventModule:Startup (Sequence Diagram)

This diagram shows the processing that is done at GUI startup. The GUI calls the startup() method of the GUITrafficEventModule, and the module adds the "Comm Log" button to the GUI toolbar. The module adds itself as a filter supporter, so that it can create the navigator filters when they are requested. A GUICommLog object is created and added to the DataModel. Also, the TrafficEventPushReceiver and CommLogPushReceiver objects are created and activated, so that they can receive any CORBA events that are pushed on either the TrafficEvent or CommLog event channels, respectively. Sometime later, the FilterManager object may call the GUITrafficEventModule to create any default system filters if the filters were not loaded successfully from the system profile. If this happens, the module will create an EventNavGroup system filter and return it.

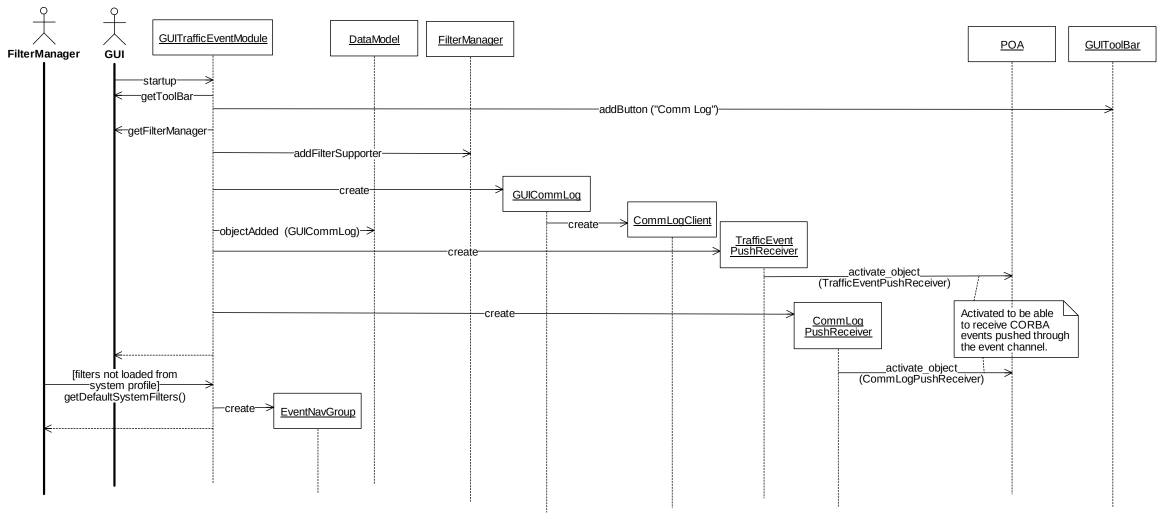


Figure 172. GUITrafficEventModule:Startup (Sequence Diagram)

3.13 GUIUserManagementModule

3.13.1 Class Diagrams

3.13.1.1 GUIUserManagementClasses (Class Diagram)

This diagram shows the classes used by the GUIUserManagement module and their relationships.

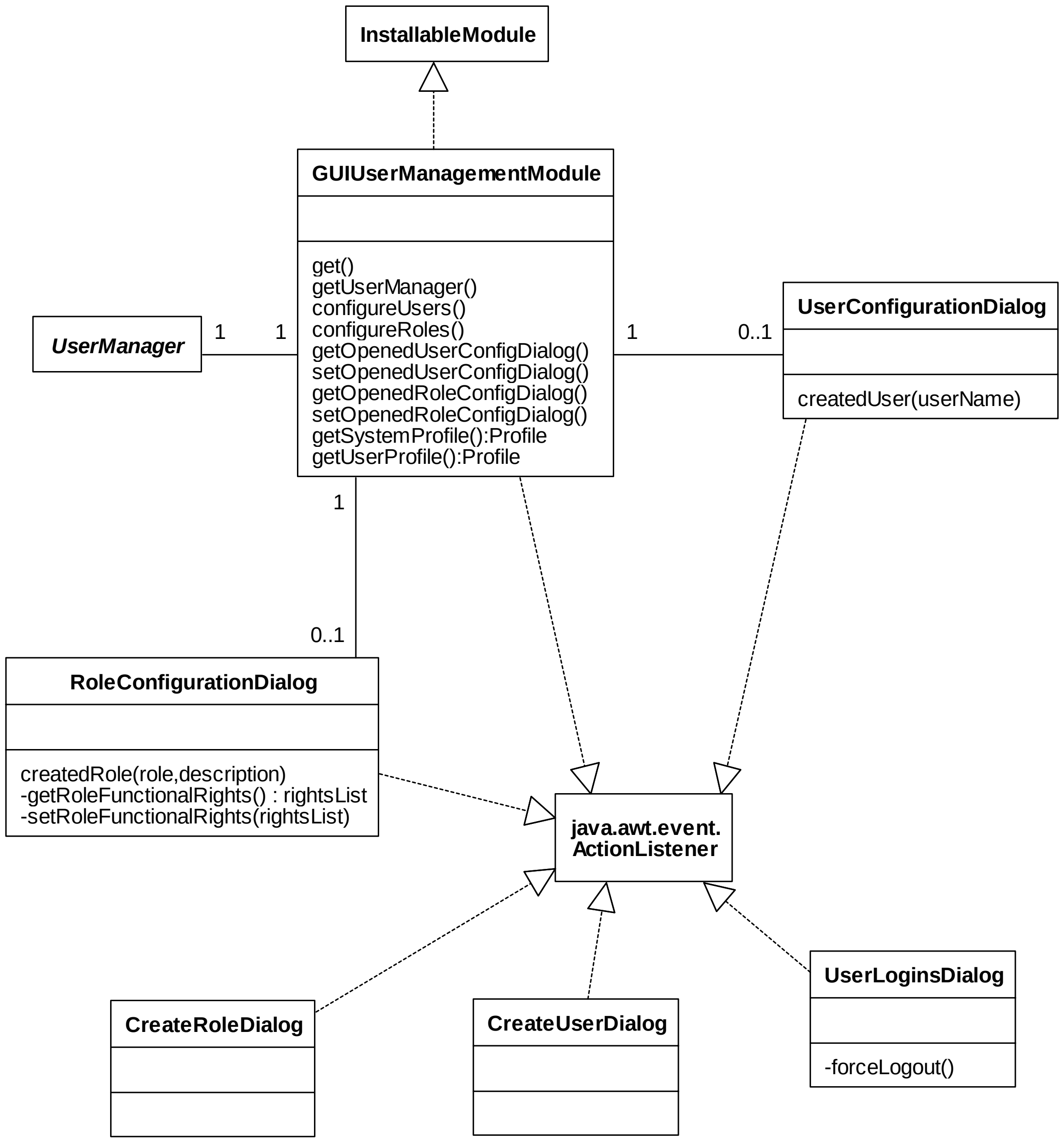


Figure 173. GUIUserManagementClasses (Class Diagram)

3.13.1.1.1 CreateRoleDialog (Class)

This dialog allows the administrator to create a new role.

3.13.1.1.2 CreateUserDialog (Class)

This dialog allows the administrator to create a new user.

3.13.1.1.3 GUIUserManagementModule (Class)

This class implements the InstallableModule interface and performs functionality for managing user rights. It can be called to configure the roles and users, or to force a logout.

3.13.1.1.4 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

3.13.1.1.5 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.13.1.1.6 RoleConfigurationDialog (Class)

This dialog allows the administrator to configure the roles in the system. It supports the Create Role, Delete Role, and Set Role Functional Rights functionality. If the user does not have role configuration rights, all editing functionality will be disabled.

3.13.1.1.7 UserConfigurationDialog (Class)

This dialog allows the administrator to view or configure the users' roles, assuming the roles have been defined. It supports the Create User, Delete User, Change User Password, Grant Role and Revoke Role functionality. If the user has view rights but not configuration rights, all configuration abilities will be disabled.

3.13.1.1.8 UserLoginsDialog (Class)

This dialog displays a list of currently logged in users and allows the administrator to force one or more users to be logged out.

3.13.1.1.9 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

3.13.2 Sequence Diagrams

3.13.2.1 GUIUserManagementModule:AddUser (Sequence Diagram)

This diagram shows how a user is added to the system. From the User Configuration Dialog, the administrator clicks on “New User”, and the Create User Dialog is invoked. When the administrator clicks “OK”, if the new password is the same as the confirmation password, the dialog will call the UserManager to create the user. If the user name or password are invalid, a message box will be displayed and the administrator will be given a chance to correct the mistake. If the user was successfully created, it will be added to the User Configuration dialog if it is still open.

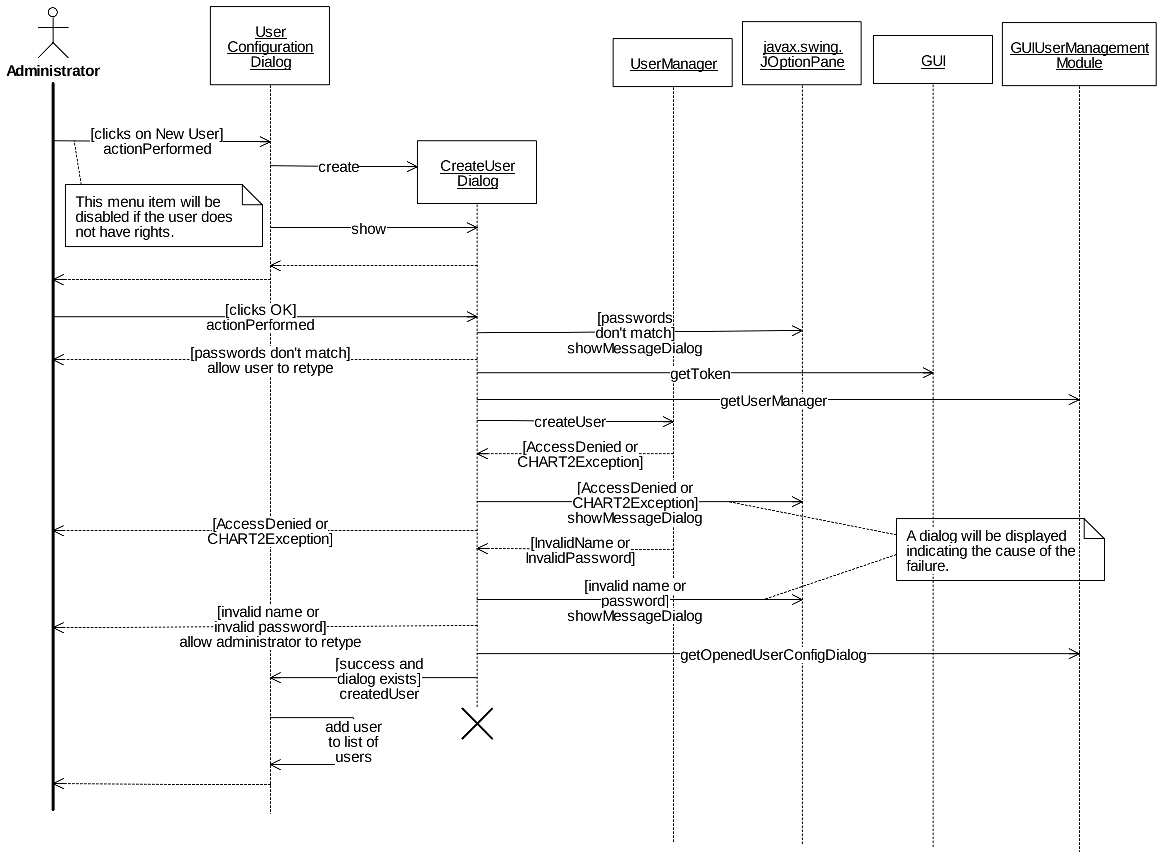

Administrator
User Configuration Dialog
UserManager
javax.swing. JOptionPane
GUI
GUIUserManagement Module
[clicks on New User] actionPerformed
create
CreateUser Dialog
This menu item will be disabled if the user does not have rights.
show
[clicks OK] actionPerformed
[passwords don't match] showMessageDialog
[passwords don't match] allow user to retype
getToken
getUserManager
createUser
[AccessDenied or CHART2Exception]
[AccessDenied or CHART2Exception] showMessageDialog
[AccessDenied or CHART2Exception]
A dialog will be displayed indicating the cause of the failure.
[InvalidName or InvalidPassword]
[invalid name or password] showMessageDialog
[invalid name or invalid password] allow administrator to retype
getOpenedUserConfigDialog
[success and dialog exists] createdUser
add user to list of users

Figure 174. GUIUserManagementModule:AddUser (Sequence Diagram)

3.13.2.2 GUIUserManagementModule:ConfigureRoles (Sequence Diagram)

This diagram shows how the Role Configuration Dialog is invoked. The adminstrator clicks on the “Configure Roles” toolbar button. The GUIUserManagementModule then creates the Role Configuration Dialog. This gets the Organizations from the trader, and gets all of the roles. It then gets the functional rights for the first role in the list. It displays the roles, functional rights within a role, and organizations supporting a given functional right. If the user does not have the ConfigureRoles right, all editing features will be disabled.

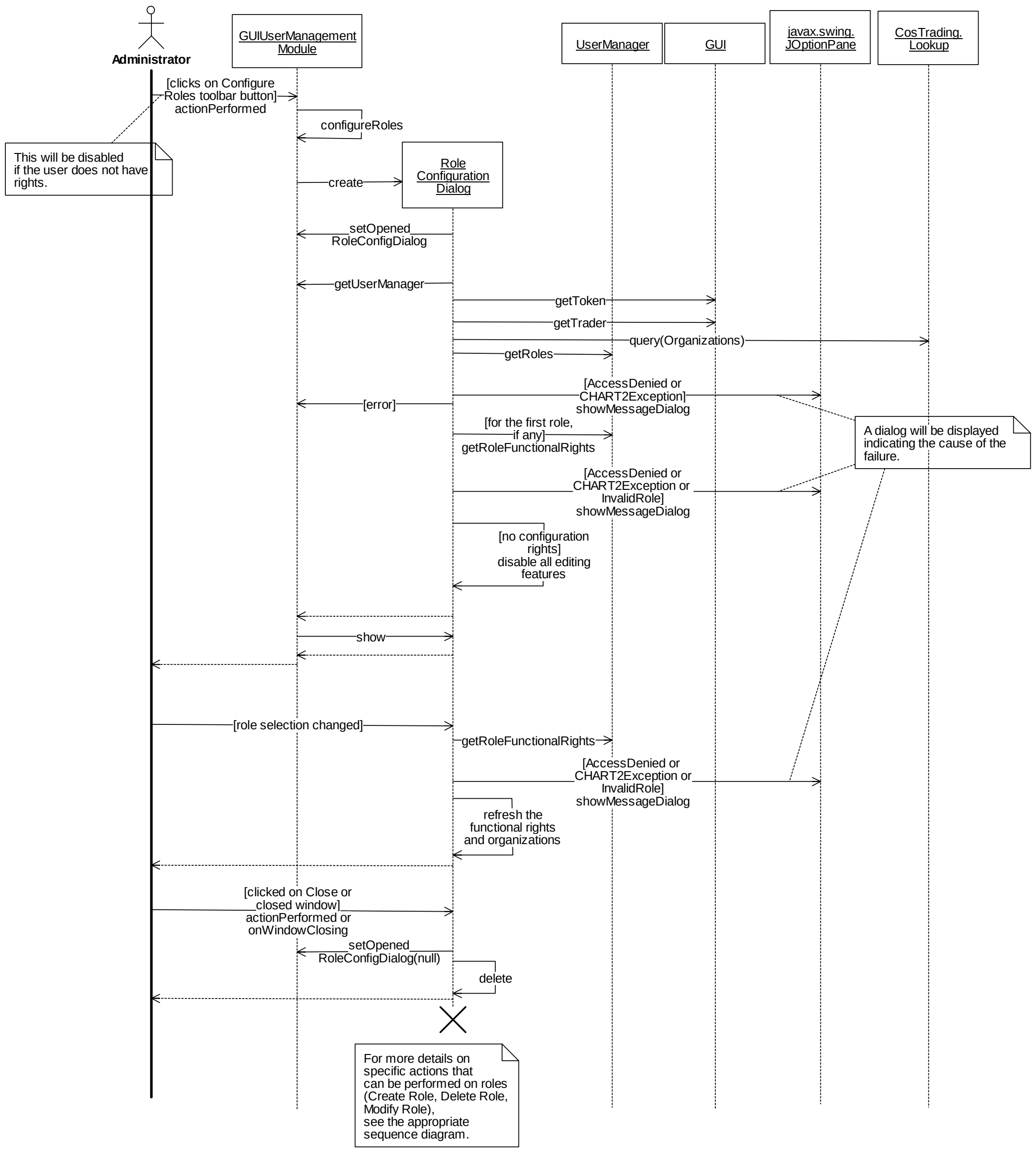

Administrator
GUIUserManagement Module
UserManager
GUI
javax.swing.JOptionPane
CosTrading.Lookup
Role Configuration Dialog
[clicks on Configure Roles toolbar button] actionPerformed
configureRoles
This will be disabled if the user does not have rights.
create
setOpened RoleConfigDialog
getUserManager
getToken
getTrader
query(Organizations)
getRoles
[AccessDenied or CHART2Exception] showMessageDialog
[error]
[for the first role, if any] getRoleFunctionalRights
A dialog will be displayed indicating the cause of the failure.
[AccessDenied or CHART2Exception or InvalidRole] showMessageDialog
[no configuration rights] disable all editing features
show
[role selection changed]
getRoleFunctionalRights
[AccessDenied or CHART2Exception or InvalidRole] showMessageDialog
refresh the functional rights and organizations
[clicked on Close or closed window] actionPerformed or onWindowClosing
setOpened RoleConfigDialog(null)
delete
For more details on specific actions that can be performed on roles (Create Role, Delete Role, Modify Role), see the appropriate sequence diagram.

Figure 175. GUIUserManagementModule:ConfigureRoles (Sequence Diagram)

3.13.2.3 GUIUserManagementModule:ConfigureUsers (Sequence Diagram)

This diagram shows how the User Configuration Dialog is invoked. The user clicks on the "Configure Users" button from the toolbar, which will be disabled unless the user has the rights: ConfigureUsers or ViewUserConfiguration. The GUIUserManagementModule will create the UserConfigurationDialog, and it will call the UserManager to get the users and the user roles. If the user has ViewUserConfiguration rights only, all user configuration functionality in the dialog will be disabled.

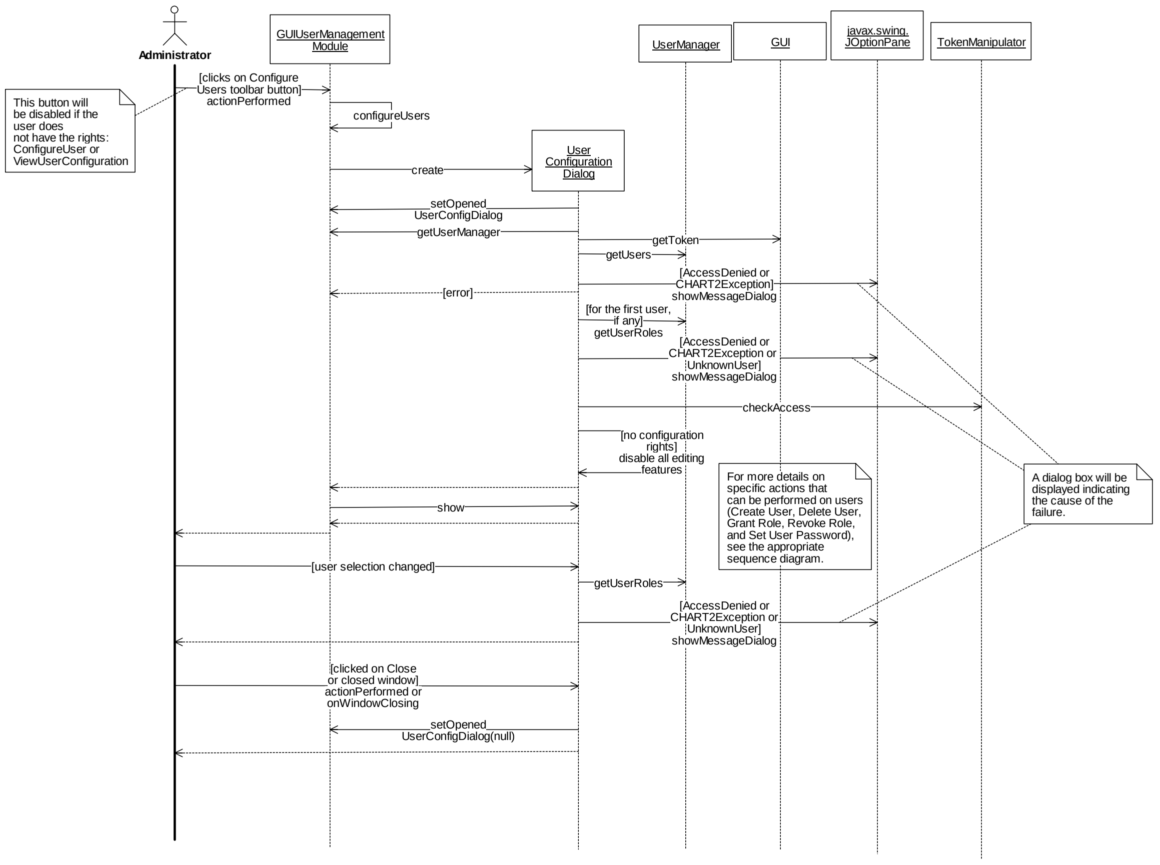
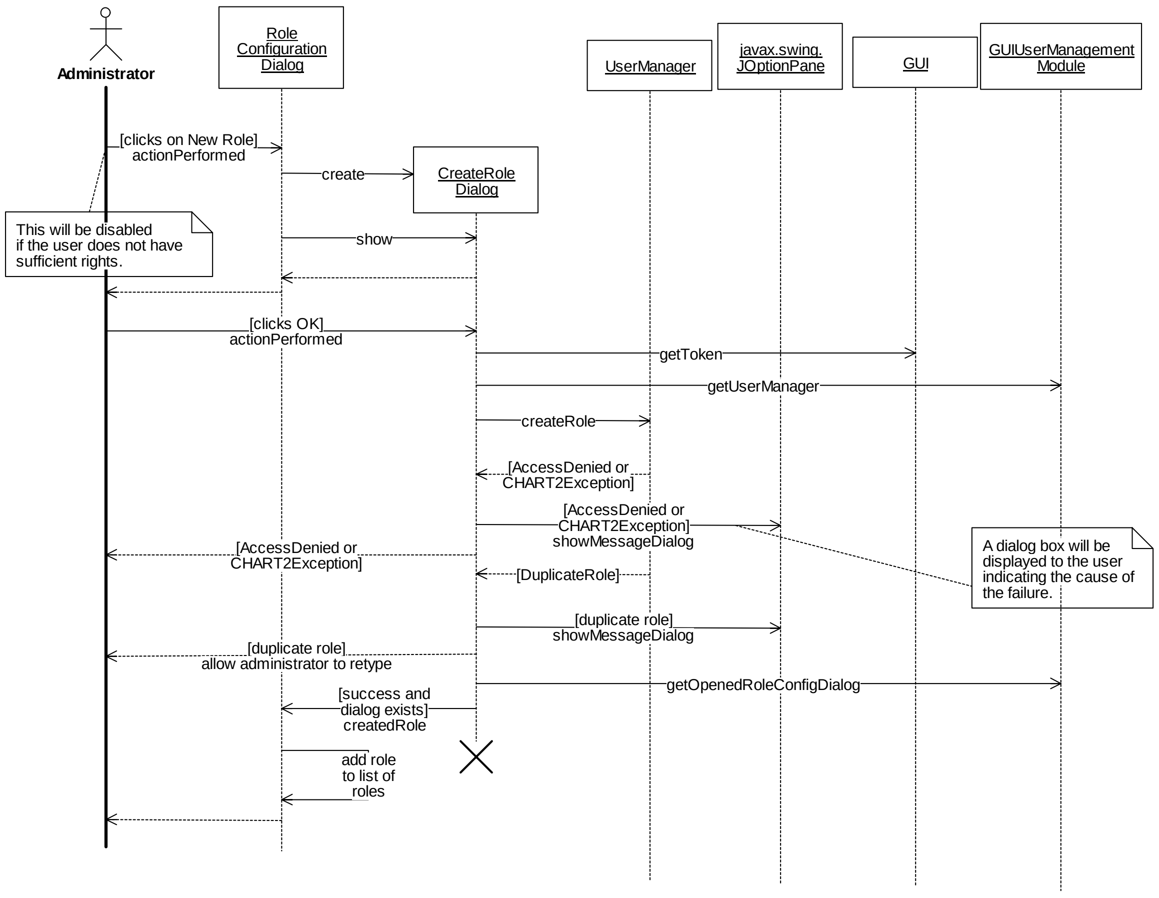

Administrator
GUIUserManagement Module
UserManager
GUI
javax.swing. JOptionPane
TokenManipulator
This button will be disabled if the user does not have the rights: ConfigureUser or ViewUserConfiguration
[clicks on Configure Users toolbar button] actionPerformed
configureUsers
create
User Configuration Dialog
setOpened UserConfigDialog
getUserManager
getToken
getUsers
[AccessDenied or CHART2Exception] showMessageDialog
[error]
[for the first user, if any] getUserRoles
[AccessDenied or CHART2Exception or UnknownUser] showMessageDialog
checkAccess
[no configuration rights] disable all editing features
For more details on specific actions that can be performed on users (Create User, Delete User, Grant Role, Revoke Role, and Set User Password), see the appropriate sequence diagram.
A dialog box will be displayed indicating the cause of the failure.
show
[user selection changed]
getUserRoles
[AccessDenied or CHART2Exception or UnknownUser] showMessageDialog
[clicked on Close or closed window] actionPerformed or onWindowClosing
setOpened UserConfigDialog(null)

Figure 176. GUIUserManagementModule:ConfigureUsers (Sequence Diagram)

3.13.2.4 GUIUserManagementModule:CreateRole (Sequence Diagram)

This diagram shows how a role is added to the system. From the Role Configuration Dialog, the administrator clicks on “New Role”, and the Create Role Dialog is invoked. When the administrator clicks “OK”, the dialog will call the UserManager to create the role. If the role is a duplicate, a message box will be displayed and the administrator will be given a chance to correct the mistake. If the role was successfully created, it will be added to the Role Configuration Dialog if it is still open.



Administrator
Role Configuration Dialog
UserManager
javax.swing.JOptionPane
GUI
GUIUserManagementModule
[clicks on New Role] actionPerformed
create
CreateRole Dialog
This will be disabled if the user does not have sufficient rights.
show
[clicks OK] actionPerformed
getToken
getUserManager
createRole
[AccessDenied or CHART2Exception]
[AccessDenied or CHART2Exception] showMessageDialog
A dialog box will be displayed to the user indicating the cause of the failure.
[AccessDenied or CHART2Exception]
[DuplicateRole]
[duplicate role] showMessageDialog
[duplicate role] allow administrator to retype
getOpenedRoleConfigDialog
[success and dialog exists] createdRole
add role to list of roles

Figure 177. GUIUserManagementModule:CreateRole (Sequence Diagram)

3.13.2.5 GUIUserManagementModule:DeleteRole (Sequence Diagram)

This diagram shows how a role is deleted from the system. From the Role Configuration Dialog, the administrator selects a role and clicks on “Delete Role”. The dialog handles the command and calls the UserManager to delete the role. If successful, the role is removed from the displayed list.

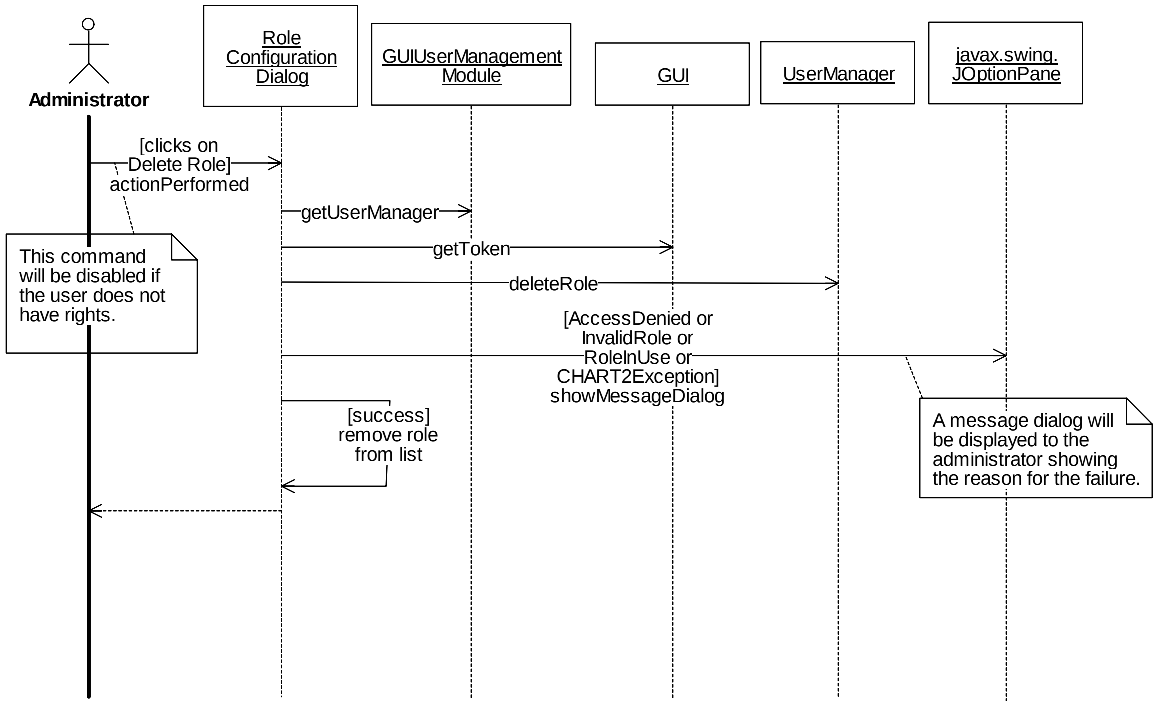

Administrator
Role Configuration Dialog
GUIUserManagement Module
GUI
UserManager
javax.swing. JOptionPane
[clicks on Delete Role] actionPerformed
getUserManager
getToken
deleteRole
[AccessDenied or InvalidRole or RoleInUse or CHART2Exception] showMessageDialog
[success] remove role from list
This command will be disabled if the user does not have rights.
A message dialog will be displayed to the administrator showing the reason for the failure.

Figure 178. GUIUserManagementModule:DeleteRole (Sequence Diagram)

3.13.2.6 GUIUserManagementModule:DeleteUser (Sequence Diagram)

This diagram shows how a user is deleted from the system. The administrator selects a user and clicks on “Delete User” from the User Configuration Dialog. The dialog calls the UserManager, which deletes the user from the system. If the user is currently logged in, a message box will be displayed informing the administrator. If the user is successfully deleted, the user’s name will be removed from the dialog.

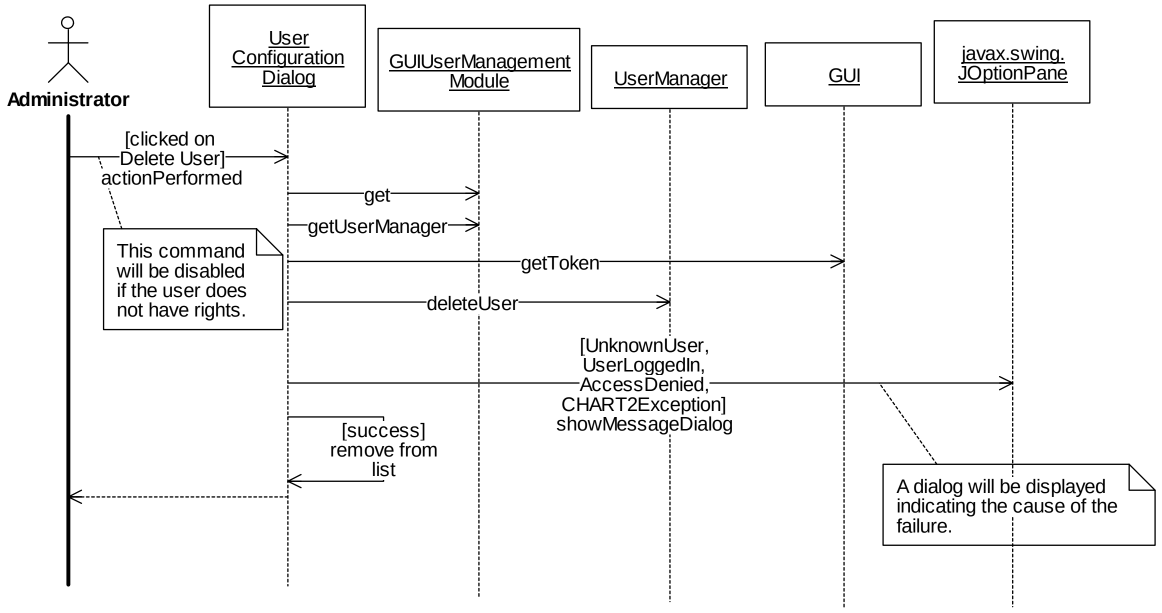

Administrator
User Configuration Dialog
GUIUserManagement Module
UserManager
GUI
javax.swing. JOptionPane
[clicked on Delete User] actionPerformed
get
getUserManager
getToken
deleteUser
This command will be disabled if the user does not have rights.
[UnknownUser, UserLoggedIn, AccessDenied, CHART2Exception] showMessageDialog
[success] remove from list
A dialog will be displayed indicating the cause of the failure.

Figure 179. GUIUserManagementModule:DeleteUser (Sequence Diagram)

3.13.2.7 GUIUserManagementModule:Discovery (Sequence Diagram)

This diagram shows how UserManager objects are discovered. The GUI will call the GUIUserManagementModule to discover objects, and the module will query the trader for any published UserManager objects. Then it will try to ping each one until one responds, and if the ping is successful, it will store the UserManager for later use. Once a UserManager is stored, it will be pinged first before querying from the trader.

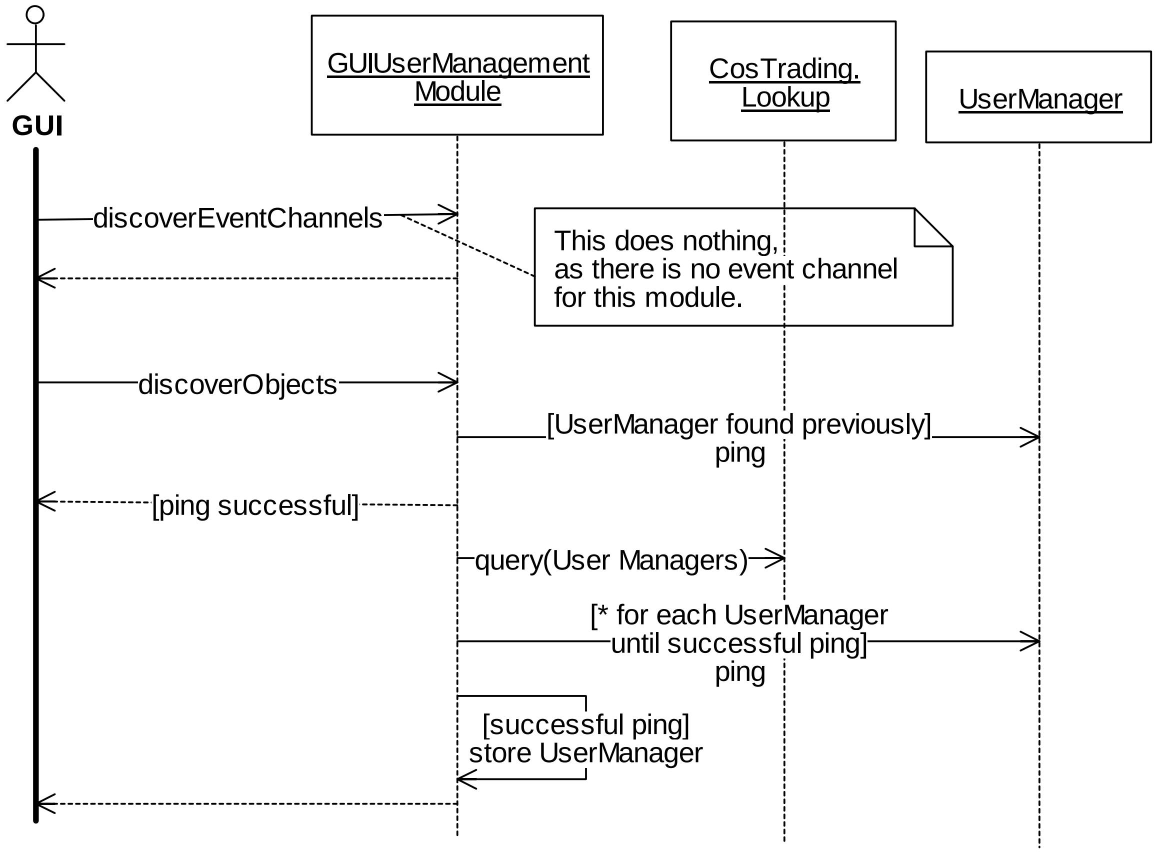

GUI
GUIUserManagement Module
CosTrading. Lookup
UserManager
discoverEventChannels
This does nothing, as there is no event channel for this module.
discoverObjects
[UserManager found previously] ping
[ping successful]
query(User Managers)
[* for each UserManager until successful ping] ping
[successful ping] store UserManager

Figure 180. GUIUserManagementModule:Discovery (Sequence Diagram)

3.13.2.8 GUIUserManagementModule:ForceLogout (Sequence Diagram)

This diagram shows how the Force Logout command is performed. The administrator clicks on the Force Logout button on the toolbar. The GUIUserManagementModule then creates a ForceLogoutDialog, which displays all of the users from all of the Operations Centers. When the adminstrator selects a user and hits the Force Logout button on the dialog, the Operations Center will be called to log the user out.

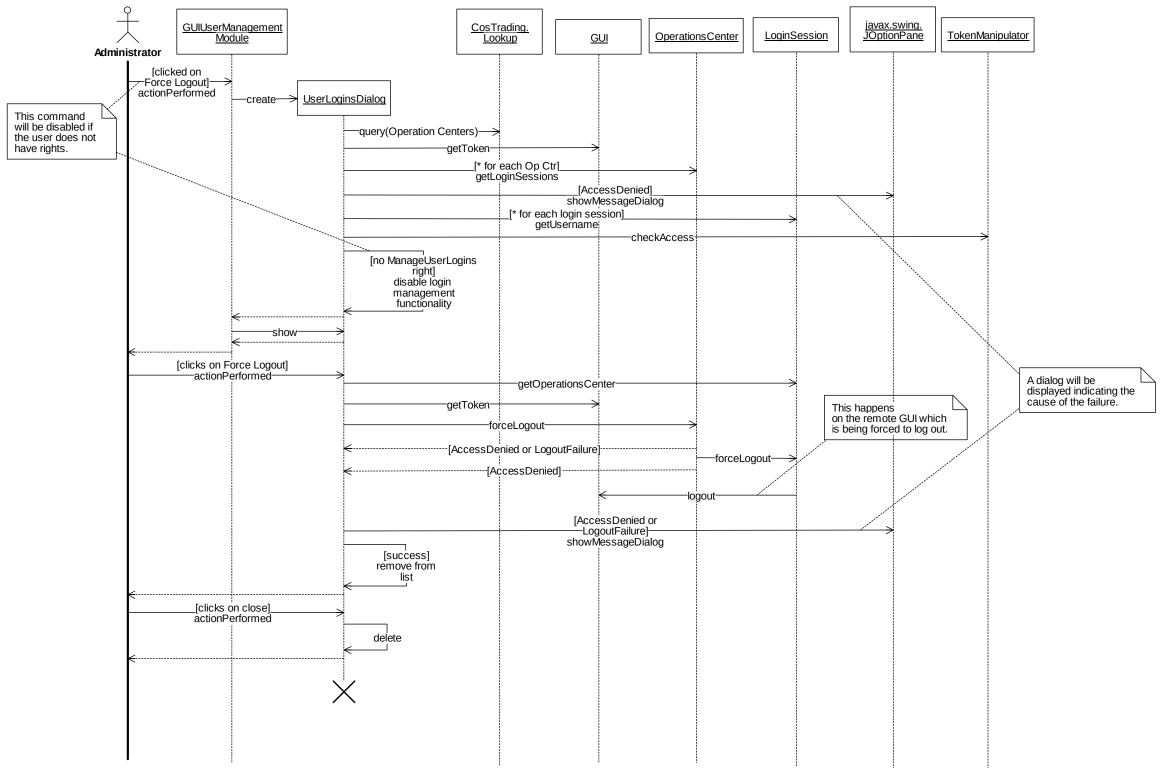

Administrator
GUIUserManagement Module
CosTrading. Lookup
GUI
OperationsCenter
LoginSession
javax.swing. JOptionPane
TokenManipulator
UserLoginsDialog
[clicked on Force Logout] actionPerformed
create
This command will be disabled if the user does not have rights.
query(Operation Centers)
getToken
[* for each Op Ctr] getLoginSessions
[AccessDenied] showMessageDialog
[* for each login session] getUsername
checkAccess
[no ManageUserLogins right] disable login management functionality
show
[clicks on Force Logout] actionPerformed
getOperationsCenter
getToken
forceLogout
[AccessDenied or LogoutFailure]
forceLogout
[AccessDenied]
logout
This happens on the remote GUI which is being forced to log out.
A dialog will be displayed indicating the cause of the failure.
[AccessDenied or LogoutFailure] showMessageDialog
[success] remove from list
[clicks on close] actionPerformed
delete

Figure 181. GUIUserManagementModule:ForceLogout (Sequence Diagram)

3.13.2.9 GUIUserManagementModule:GrantRole (Sequence Diagram)

This diagram shows how a role is granted to a user. From the User Configuration dialog, the administrator clicks on an (unchecked) role checkbox in the role list. The dialog will mark the role as checked, which assumes a successful operation. Then it will call the UserManager to grant the role. On failure, a message box will be displayed and the role will be unchecked.

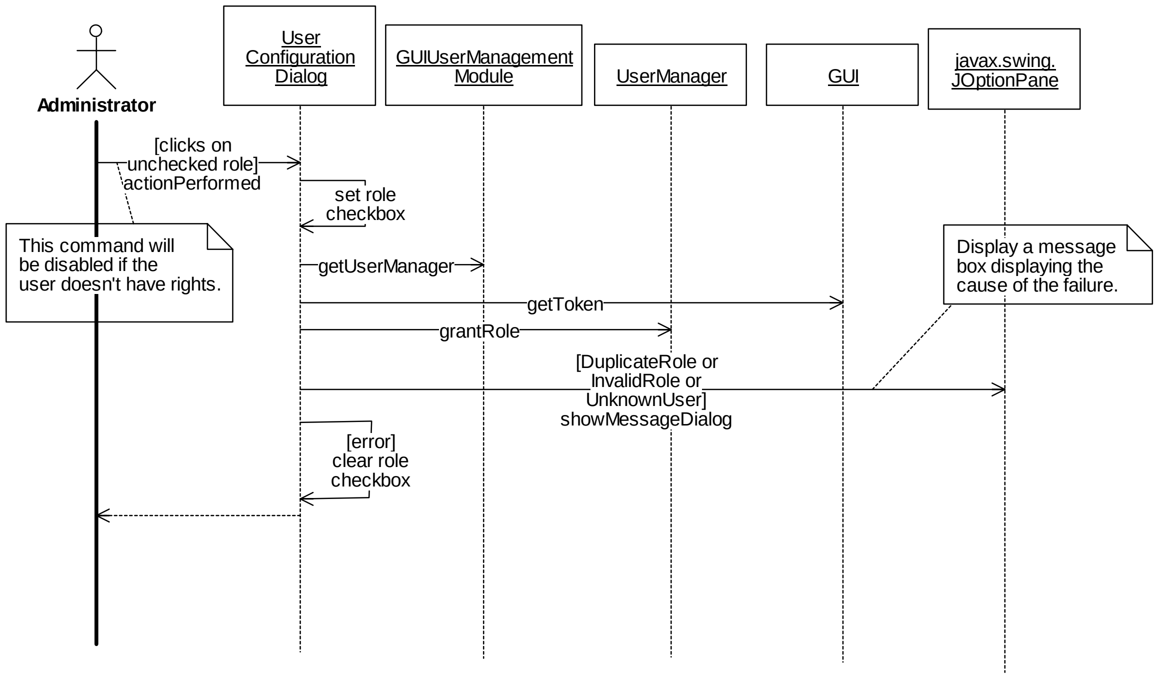

Administrator
User Configuration Dialog
GUIUserManagement Module
UserManager
GUI
javax.swing. JOptionPane
[clicks on unchecked role] actionPerformed
set role checkbox
This command will be disabled if the user doesn't have rights.
getUserManager
getToken
grantRole
Display a message box displaying the cause of the failure.
[DuplicateRole or InvalidRole or UnknownUser] showMessageDialog
[error] clear role checkbox

Figure 182. GUIUserManagementModule:GrantRole (Sequence Diagram)

3.13.2.10 GUIUserManagementModule:Login (Sequence Diagram)

This diagram shows the user-specific initialization that is done at login.

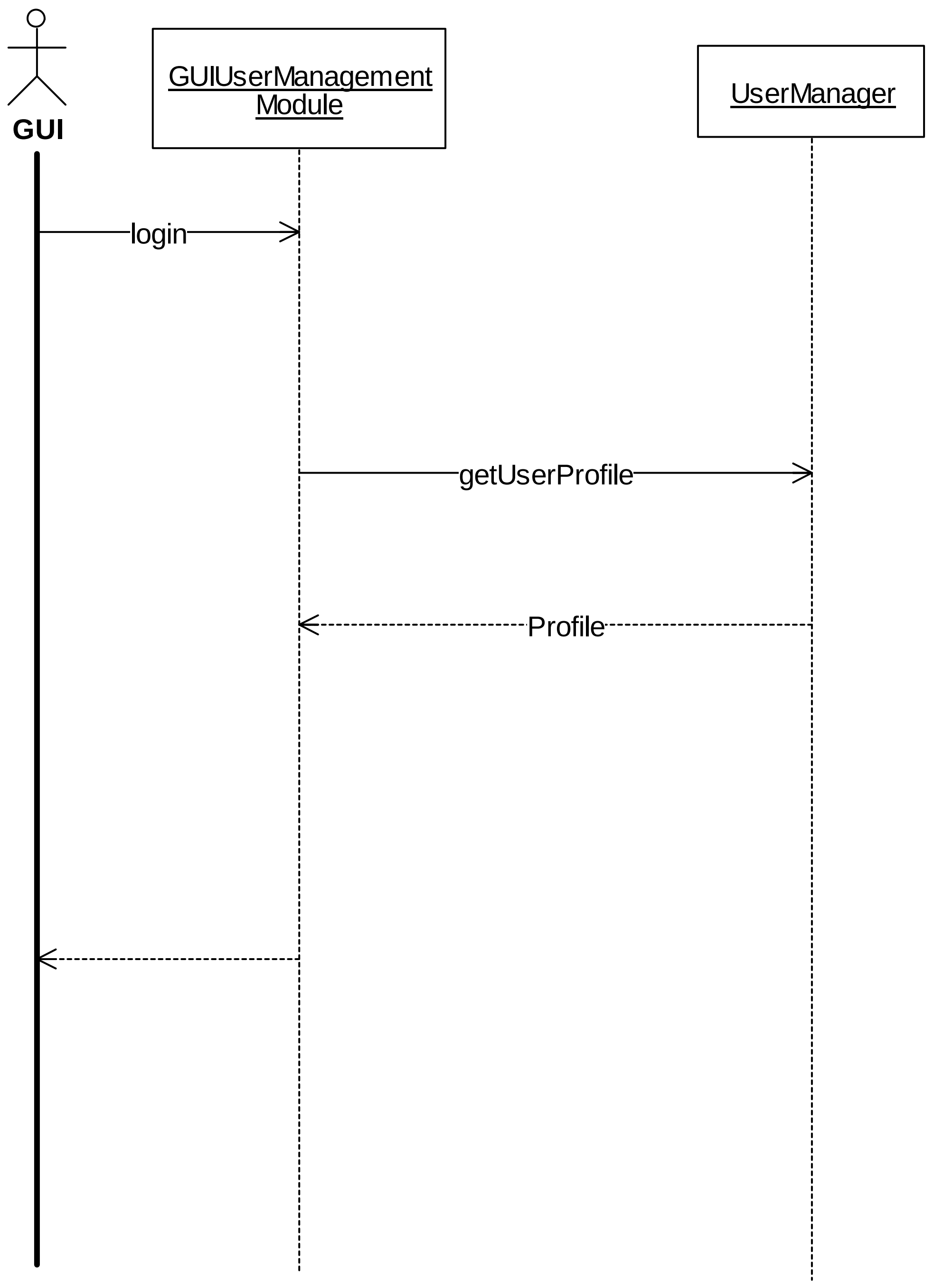


Figure 183. GUIUserManagementModule:Login (Sequence Diagram)

3.13.2.11 GUIUserManagementModule:ModifyRole (Sequence Diagram)

This diagram shows how roles are modified in the system. From the RoleConfigurationDialog, the adminstrator clicks on a functional right or an organization to toggle its presence in the role. The dialog retrieves all of the functional rights from its components, then sets the functional rights by calling the User Manager. If an error occurs, the correct functional rights for the role are retrieved from the User Manager, and the dialog is refreshed based on the correct rights.

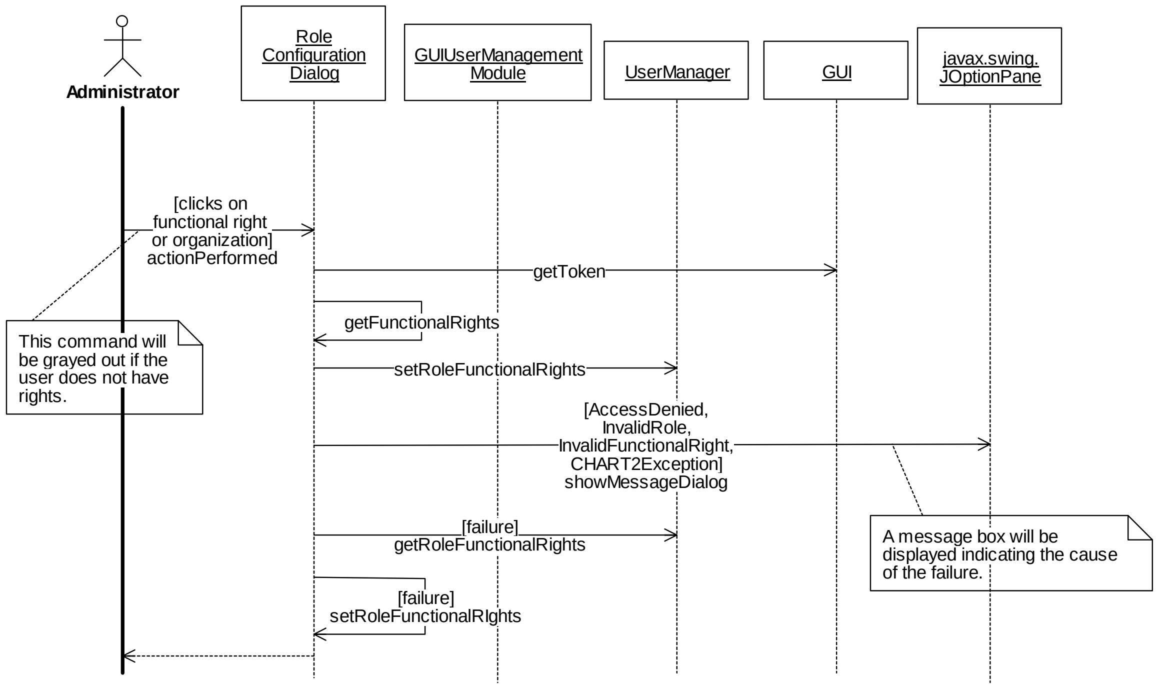

Administrator
Role Configuration Dialog
GUIUserManagement Module
UserManager
GUI
javax.swing. JOptionPane
[clicks on functional right or organization] actionPerformed
getToken
getFunctionalRights
setRoleFunctionalRights
[AccessDenied, InvalidRole, InvalidFunctionalRight, CHART2Exception] showMessageDialog
[failure] getRoleFunctionalRights
[failure] setRoleFunctionalRIghts
This command will be grayed out if the user does not have rights.
A message box will be displayed indicating the cause of the failure.

Figure 184. GUIUserManagementModule:ModifyRole (Sequence Diagram)

3.13.2.12 GUIUserManagementModule:RevokeRole (Sequence Diagram)

This diagram shows how a role is revoked from a user. From the User Configuration dialog, the administrator clicks on a (checked) role checkbox in the role list. The dialog will mark the role as unchecked, which assumes a successful operation. Then it will call the UserManager to revoke the role. On failure, a message box will be displayed and the role will be checked.

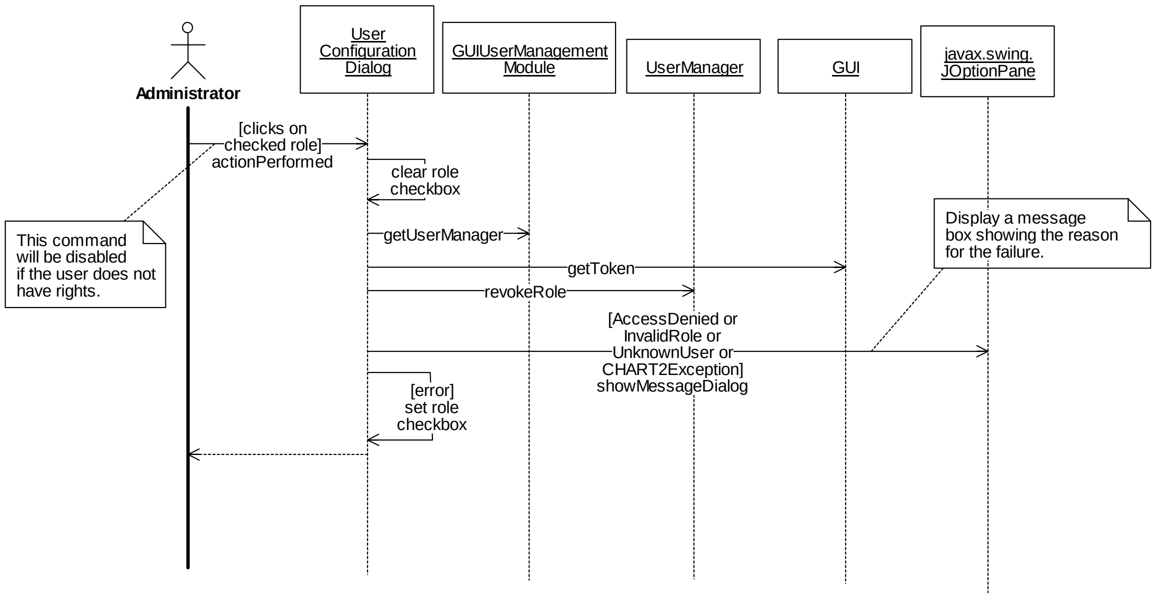

Administrator
User Configuration Dialog
GUIUserManagement Module
UserManager
GUI
javax.swing. JOptionPane
[clicks on checked role] actionPerformed
clear role checkbox
getUserManager
getToken
revokeRole
[AccessDenied or InvalidRole or UnknownUser or CHART2Exception] showMessageDialog
[error] set role checkbox
This command will be disabled if the user does not have rights.
Display a message box showing the reason for the failure.

Figure 185. GUIUserManagementModule:RevokeRole (Sequence Diagram)

3.13.2.13 GUIUserManagementModule:Startup (Sequence Diagram)

This diagram shows the actions performed by the GUIUserManagementModule at startup.

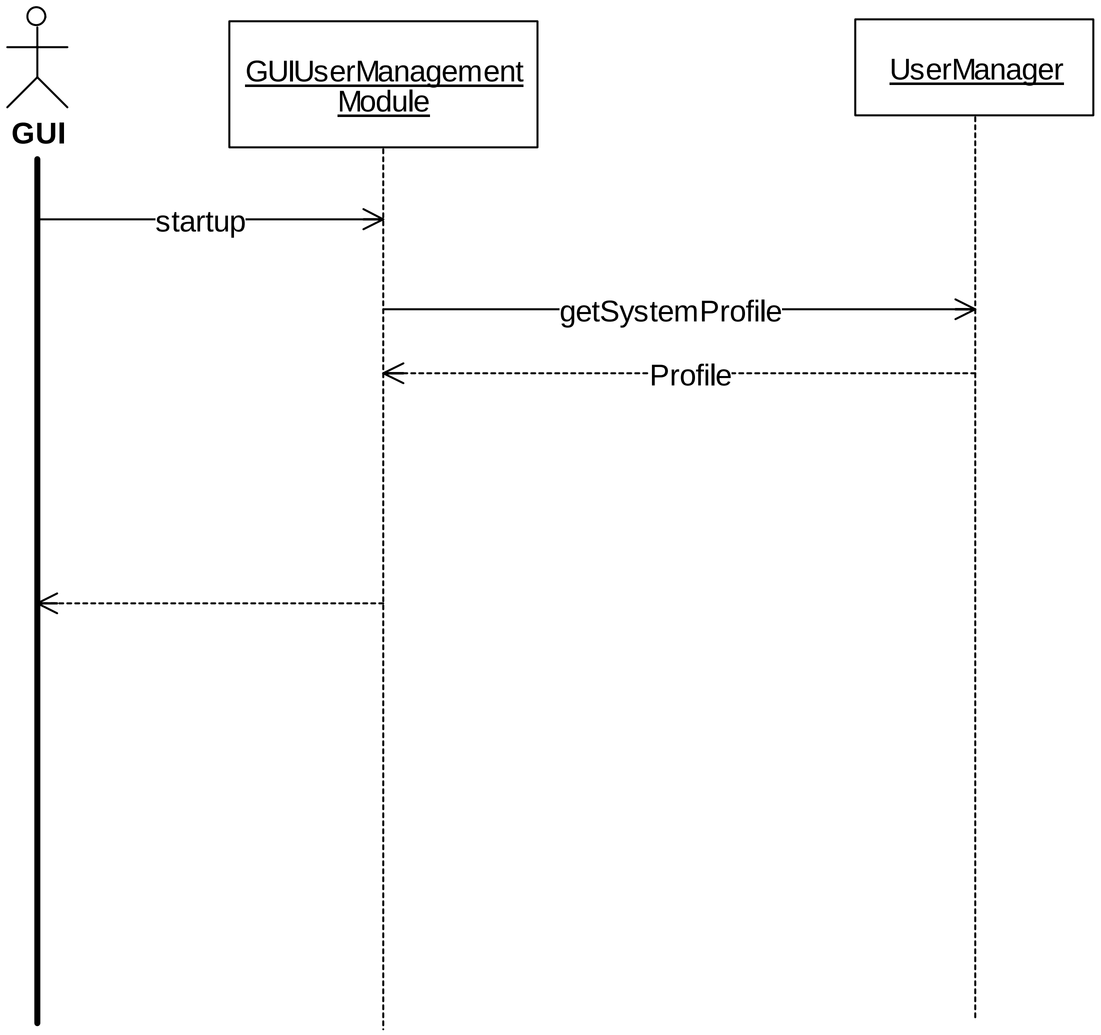


Figure 186. GUIUserManagementModule:Startup (Sequence Diagram)

3.14 GUIUtility

3.14.1 Class Diagrams

3.14.1.1 AudioClasses (Class Diagram)

This diagram shows the classes used to play audio in the GUI.

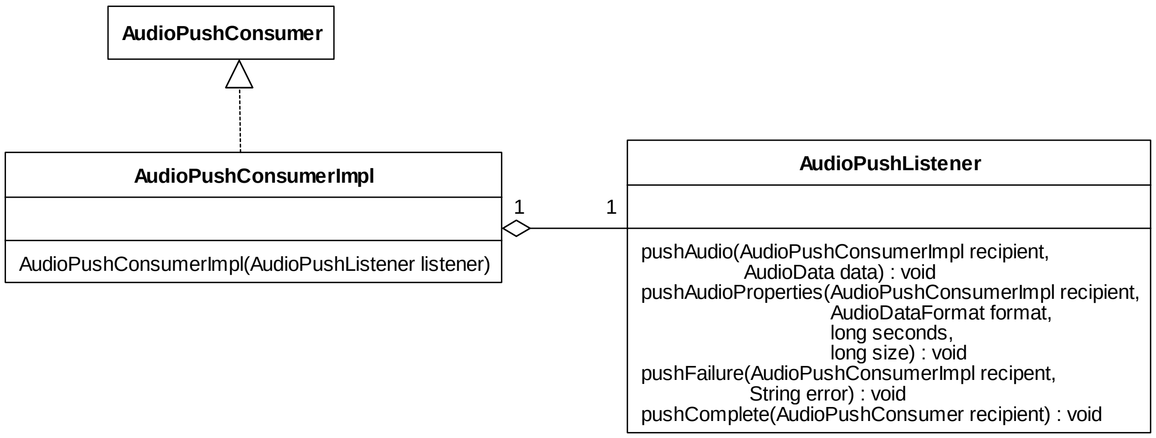

AudioPushConsumer
AudioPushConsumerImpl
AudioPushConsumerImpl(AudioPushListener listener)
1
1
AudioPushListener
pushAudio(AudioPushConsumerImpl recipient, AudioData data) : void
pushAudioProperties(AudioPushConsumerImpl recipient, AudioDataFormat format, long seconds, long size) : void
pushFailure(AudioPushConsumerImpl recipent, String error) : void
pushComplete(AudioPushConsumer recipient) : void

Figure 187. AudioClasses (Class Diagram)

3.14.1.1.1 AudioPushConsumer (Class)

This interface is implemented by objects that wish to receive audio data using the push model, where the server pushes the data to the consumer. One call to pushAudioProperties() will always precede any calls to pushAudio().

3.14.1.1.2 AudioPushConsumerImpl (Class)

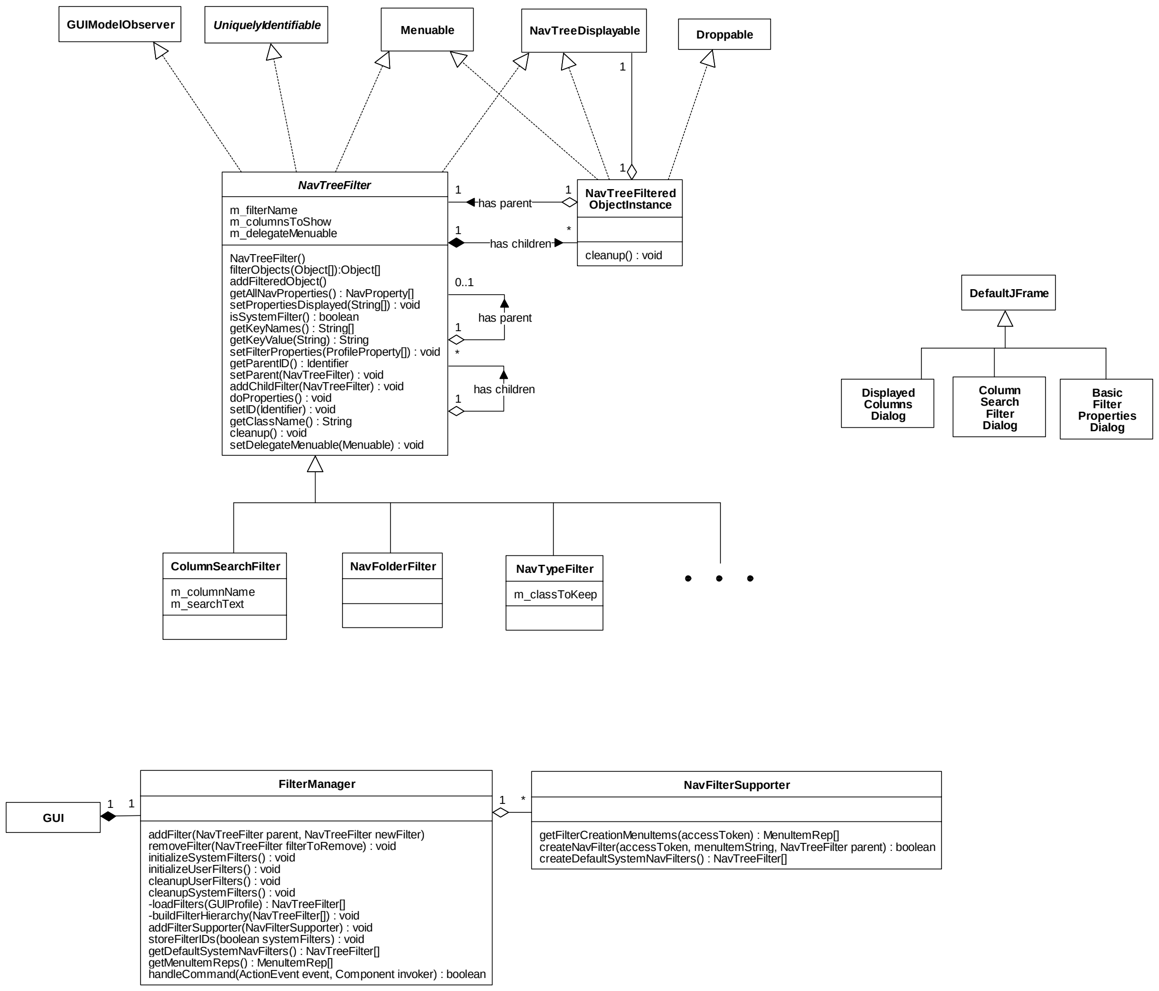
This class implements the AudioPushConsumer CORBA interface and delegates the calls to a AudioPushListener interface.

3.14.1.1.3 AudioPushListener (Class)

This is called by one or more AudioPushConsumerImpls when an audio clip is being pushed.

3.14.1.2 FilterClasses (Class Diagram)

This diagram shows the classes that are used to implement Navigator filters in the GUI. The filters are configurable via the System and User profiles and their purpose is to display a subset of the available objects in each level of the Navigator tree. Child filters return a subset of the subset of objects already filtered by the parent.



GUIModelObserver
UniquelyIdentifiable
Menuable
NavTreeDisplayable
Droppable
1
1
NavTreeFilter
m_filterName
m_columnsToShow
m_delegateMenuable
NavTreeFilter()
filterObjects(Object[]):Object[]
addFilteredObject()
getAllNavProperties() : NavProperty[]
setPropertiesDisplayed(String[]) : void
isSystemFilter() : boolean
getKeyNames() : String[]
getKeyValue(String) : String
setFilterProperties(ProfileProperty[]) : void
getParentID() : Identifier
setParent(NavTreeFilter) : void
addChildFilter(NavTreeFilter) : void
doProperties() : void
setID(Identifier) : void
getClassName() : String
cleanup() : void
setDelegateMenuable(Menuable) : void
1
has parent
1
1
has children
*
NavTreeFiltered ObjectInstance
cleanup() : void
0..1
has parent
1
*
has children
1
DefaultJFrame
Displayed Columns Dialog
Column Search Filter Dialog
Basic Filter Properties Dialog
ColumnSearchFilter
m_columnName
m_searchText
NavFolderFilter
NavTypeFilter
m_classToKeep
• • •
GUI
1
1
FilterManager
addFilter(NavTreeFilter parent, NavTreeFilter newFilter)
removeFilter(NavTreeFilter filterToRemove) : void
initializeSystemFilters() : void
initializeUserFilters() : void
cleanupUserFilters() : void
cleanupSystemFilters() : void
-loadFilters(GUIProfile) : NavTreeFilter[]
-buildFilterHierarchy(NavTreeFilter[]) : void
addFilterSupporter(NavFilterSupporter) : void
storeFilterIDs(boolean systemFilters) : void
getDefaultSystemNavFilters() : NavTreeFilter[]
getMenuItemReps() : MenuItemRep[]
handleCommand(ActionEvent event, Component invoker) : boolean
1
*
NavFilterSupporter
getFilterCreationMenuItems(accessToken) : MenuItemRep[]
createNavFilter(accessToken, menuItemString, NavTreeFilter parent) : boolean
createDefaultSystemNavFilters() : NavTreeFilter[]

Figure 188. FilterClasses (Class Diagram)

3.14.1.2.1 Basic Filter Properties Dialog (Class)

This dialog allows a user with rights to edit the properties of the filter (most notably the name).

3.14.1.2.2 Column Search Filter Dialog (Class)

This dialog allows the user to edit the filter characteristics for the PropertySearchFilter, which does a text search for columns displayed in the Navigator.

3.14.1.2.3 Displayed Columns Dialog (Class)

This dialog allows the filter to be edited so that each of the columns displayed on the right-hand side of the Navigator can be toggled on or off.

3.14.1.2.4 Droppable (Class)

This interface must be implemented by any object wishing to take part in a drag and drop operation. It is used by the DropHandler class to determine if a drop action should be allowed and to delegate the handling of the drop action after it is performed.

3.14.1.2.5 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

3.14.1.2.6 ColumnSearchFilter (Class)

This filter will show any objects whose text value listed in the specified navigator column contains the specified text.

3.14.1.2.7 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.14.1.2.8 DefaultJFrame (Class)

This class provides a default implementation of the WindowManageable interface, and may be used as a base class for other frame windows in the GUI. It handles all interactions with the WindowManager for attaching and detaching, as well as saving the window position.

3.14.1.2.9 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

3.14.1.2.10 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

3.14.1.2.11 NavTreeFilter (Class)

This class serves as a node in the Navigator tree and filters objects to be displayed in the Navigator. It is an observer to the DataModel so that it can create the NavTreeFilteredObjectInstance objects for any NavTreeDisplayables that it contains. (Multiple instances can appear to represent one NavTreeDisplayable object). Filters can be cascaded to achieve a cumulative filtering effect; that is, a filter appearing under a parent filter will call the parent filter first to filter the objects, and then it will apply its own filtering method. The cascading of filters is therefore an "AND" operation. A filter can either be a system filter or a user-specific filter. System filters can only be modified by someone with the correct administrative rights, and they can only be added as a child of other system filters.

3.14.1.2.12 NavTreeFiltered ObjectInstance (Class)

This class represents an instance of an object which is displayed under a filter. The object being represented is a NavTreeDisplayable which is neither a NavTreeFilter nor a NavTreeFilteredObjectInstance, and passes through all of the filters from the root up to the filter containing this representation. There can be more than one instance of the wrapped object appearing in the Navigator tree at a given time, under different branches. This object will delegate all GUI functionality to the object that it represents. The filter will watch the DataModel to determine when objects are eligible to be displayed under the filter, at which time it will create a NavTreeFilteredObjectInstance and add it to the DataModel. The NavigatorDriver will then add the instance to the Navigator tree. Other NavTreeFilter objects and NavTreeFilteredObjectInstances will ignore the new instance by checking its type.

3.14.1.2.13 FilterManager (Class)

This class provides functionality for managing the filters in the system. As it deals with the singleton GUI and the DataModel objects, it too will be a singleton object. The GUI will create and hold the FilterManager. Filter supporters can be added to the FilterManager to support the creation of supporter-specific filter types.

3.14.1.2.14 NavFilterSupporter (Class)

This interface is used to allow type-specific filters to be created by external classes such as the installable modules. It is called to get the menu items for filter creation, as well as to create the filter when those menu items are clicked on. It is also called to provide default system filters for “bootstrapping” the system filters in case the system filters are not loaded from the system profile.

3.14.1.2.15 NavFolderFilter (Class)

This is a placeholder filter/folder whose purpose is to act as a parent for other filters. This filter will not filter out any objects, so it does not act as a filter at all.

3.14.1.2.16 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.14.1.2.17 NavTypeFilter (Class)

This filter ignores all objects that are not assignable to a given class or interface. Thus, an interface or base class can be specified and all of the objects implementing the interface or extending the base class will be included.

3.14.2 Sequence Diagrams

3.14.2.1 GUIUtility:AddFilter (Sequence Diagram)

This diagram shows how filters are added to the system. To add a filter, the user must click on an existing folder. The FilterManager will then call all of the filter supporters to get the filter creation menu items. If the filter being clicked on is a system filter, either system or user filters may be added as children. If the filter is a user filter, only user filters may be added as children. The menu items will also be grayed out if the user does not have rights to add the appropriate type of filter. After the user clicks on a menu item, the FilterManager calls each of the filter supporters to create a NavTreeFilter. If the supporter recognizes the menu item string, it will create the appropriate type of NavTreeFilter and open the properties dialog corresponding to the filter type. When the user clicks “OK”, the FilterManager will be called to add the filter to the system. This will add the NavTreeFilter to the DataModel, and it will also persist the filter properties into the system or user profile (as appropriate) so that the filter can be reconstructed the next time the GUI is started (for system filters) or the user logs in (for user filters). (See the sequence diagram UpdateForFilterChange for more details).

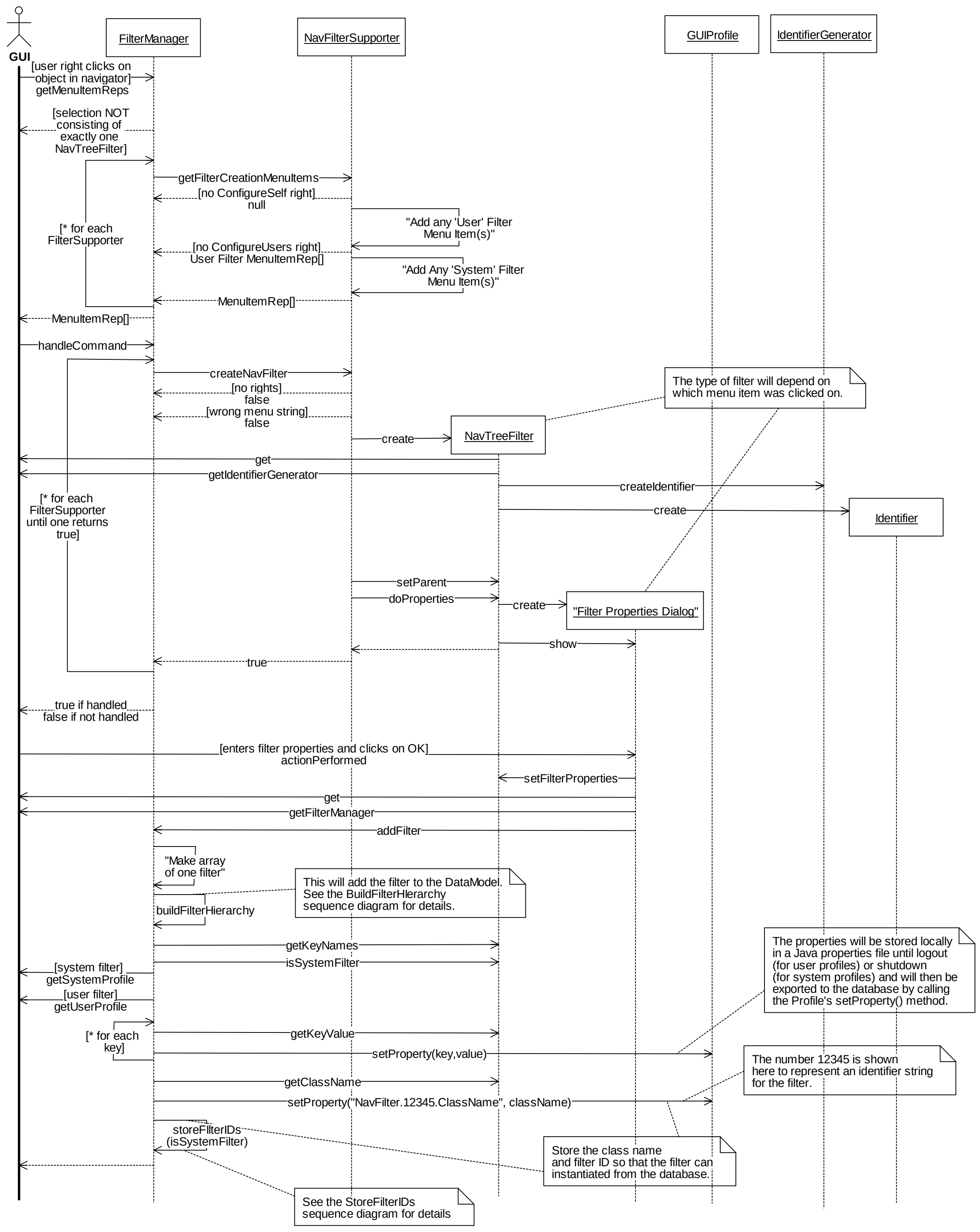

GUI
FilterManager
NavFilterSupporter
GUIProfile
IdentifierGenerator
[user right clicks on object in navigator] getMenuItemReps
[selection NOT consisting of exactly one NavTreeFilter]
getFilterCreationMenuItems
[no ConfigureSelf right] null
"Add any 'User' Filter Menu Item(s)"
[* for each FilterSupporter
[no ConfigureUsers right] User Filter MenuItemRep[]
"Add Any 'System' Filter Menu Item(s)"
MenuItemRep[]
MenuItemRep[]
handleCommand
createNavFilter
[no rights] false
[wrong menu string] false
create
NavTreeFilter
The type of filter will depend on which menu item was clicked on.
get
getIdentifierGenerator
createIdentifier
create
Identifier
[* for each FilterSupporter until one returns true]
setParent
doProperties
create
"Filter Properties Dialog"
show
true
true if handled false if not handled
[enters filter properties and clicks on OK] actionPerformed
setFilterProperties
get
getFilterManager
addFilter
"Make array of one filter"
This will add the filter to the DataModel. See the BuildFilterHIerarchy sequence diagram for details.
buildFilterHierarchy
getKeyNames
isSystemFilter
The properties will be stored locally in a Java properties file until logout (for user profiles) or shutdown (for system profiles) and will then be exported to the database by calling the Profile's setProperty() method.
[system filter] getSystemProfile
[user filter] getUserProfile
[* for each key]
getKeyValue
setProperty(key,value)
The number 12345 is shown here to represent an identifier string for the filter.
getClassName
setProperty("NavFilter.12345.ClassName", className)
storeFilterIDs (isSystemFilter)
Store the class name and filter ID so that the filter can instantiated from the database.
See the StoreFilterIDs sequence diagram for details

Figure 189. GUIUtility:AddFilter (Sequence Diagram)

3.14.2.2 GUIUtility:BuildFilterHierarchy (Sequence Diagram)

This diagram shows the building of the filter hierarchy in memory, given an existing hierarchy and some new filters to be added. It may make several passes through the filters to be added, each time searching for existing parent filters to add the new filters to. It will set up the parent/child relationship and add the filter to the DataModel. It will also attach the filter to the DataModel as an observer so that it will know when new NavTreeFilteredObjectInstance objects are to be added or removed from the filter.

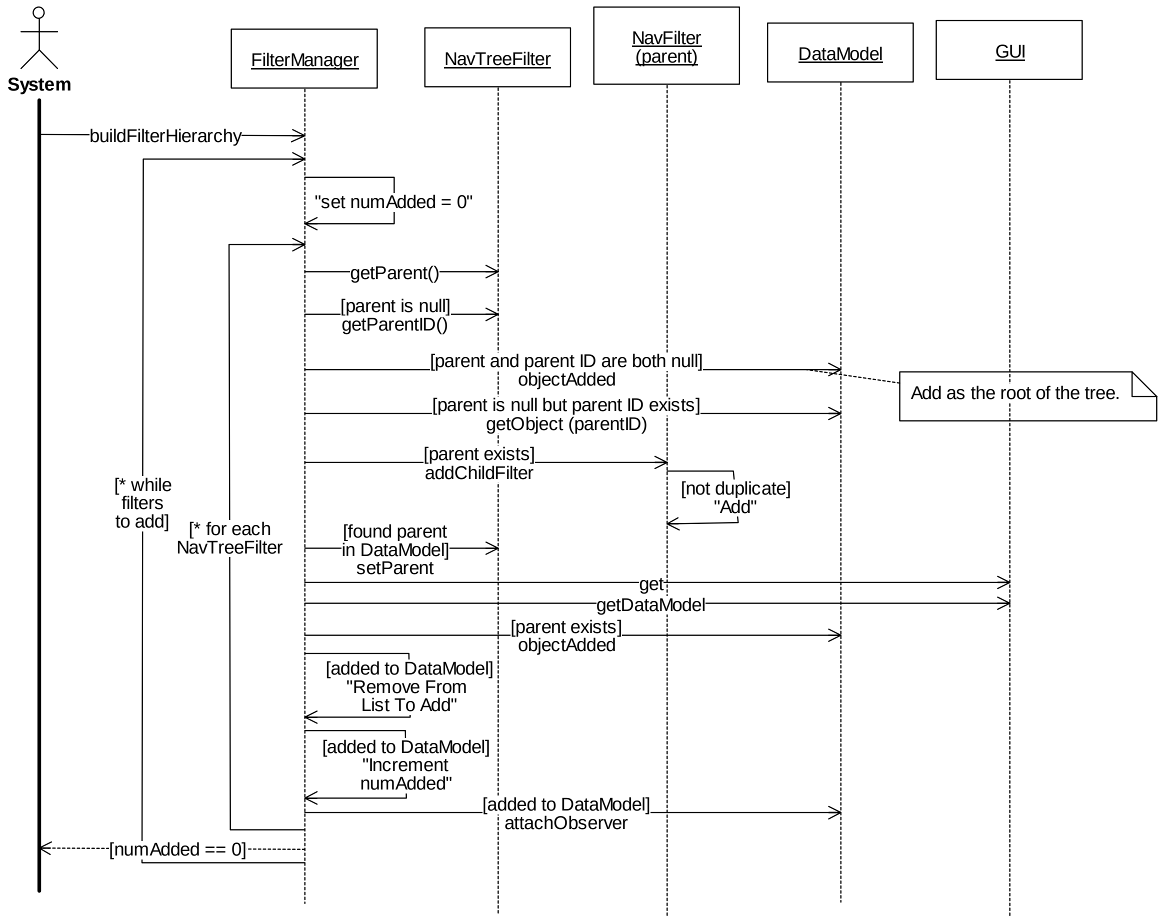

System
FilterManager
NavTreeFilter
NavFilter (parent)
DataModel
GUI
buildFilterHierarchy
"set numAdded = 0"
getParent()
[parent is null] getParentID()
[parent and parent ID are both null] objectAdded
Add as the root of the tree.
[parent is null but parent ID exists] getObject (parentID)
[parent exists] addChildFilter
[not duplicate] "Add"
[* while filters to add]
[* for each NavTreeFilter
[found parent in DataModel] setParent
get
getDataModel
[parent exists] objectAdded
[added to DataModel] "Remove From List To Add"
[added to DataModel] "Increment numAdded"
[added to DataModel] attachObserver
[numAdded == 0]

Figure 190. GUIUtility:BuildFilterHierarchy (Sequence Diagram)

3.14.2.3 GUIUtility:CleanupSystemFilters (Sequence Diagram)

This diagram shows the cleanup of the system filters at shutdown. The GUI calls the FilterManager, which gets all of the NavTreeFilter objects from the DataModel and calls cleanup on each, thereafter removing them from the DataModel. Each filter detaches itself from the DataModel as an observer. The filter also cleans up any NavTreeFilteredObjectInstance objects that it contains.

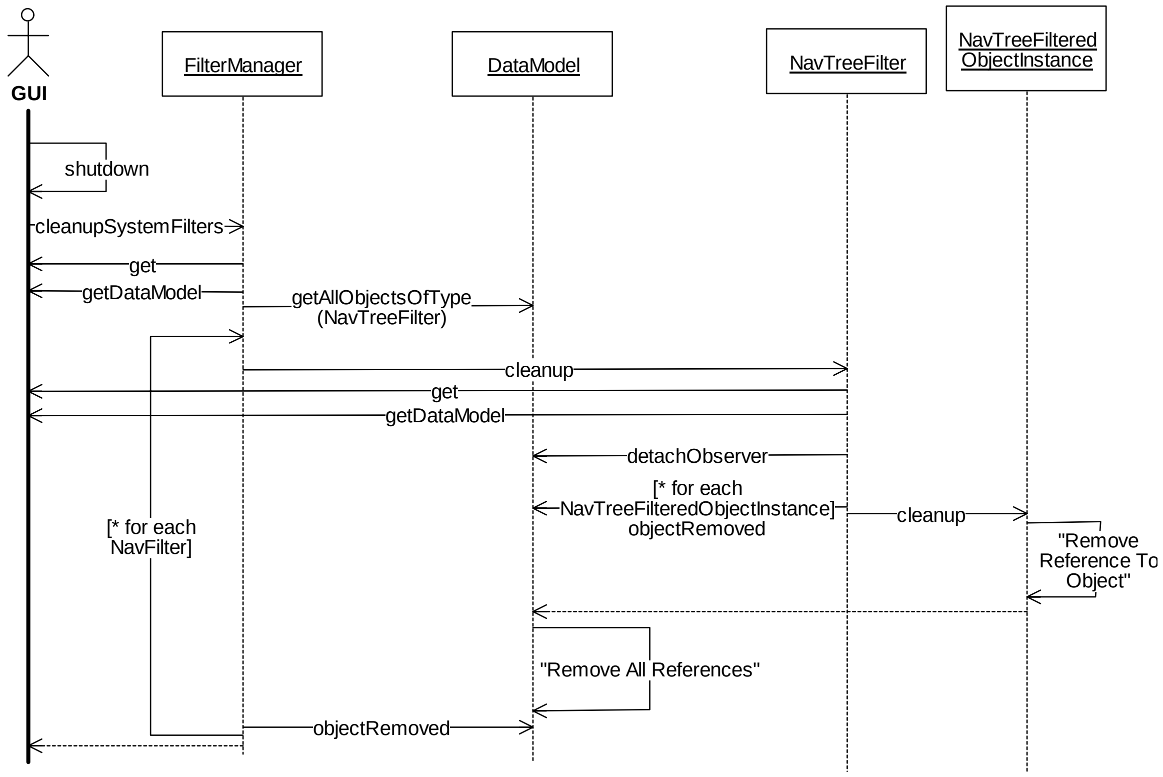


Figure 191. GUIUtility:CleanupSystemFilters (Sequence Diagram)

3.14.2.4 GUIUtility:CleanupUserFilters (Sequence Diagram)

This diagram shows the cleanup of the user filters at logout. The GUI calls the FilterManager, which gets all of the NavTreeFilter objects from the DataModel which are user filters and calls cleanup on each, thereafter removing them from the DataModel. Each filter detaches itself from the DataModel as an observer. The filter also cleans up any NavTreeFilteredObjectInstance objects that it contains.

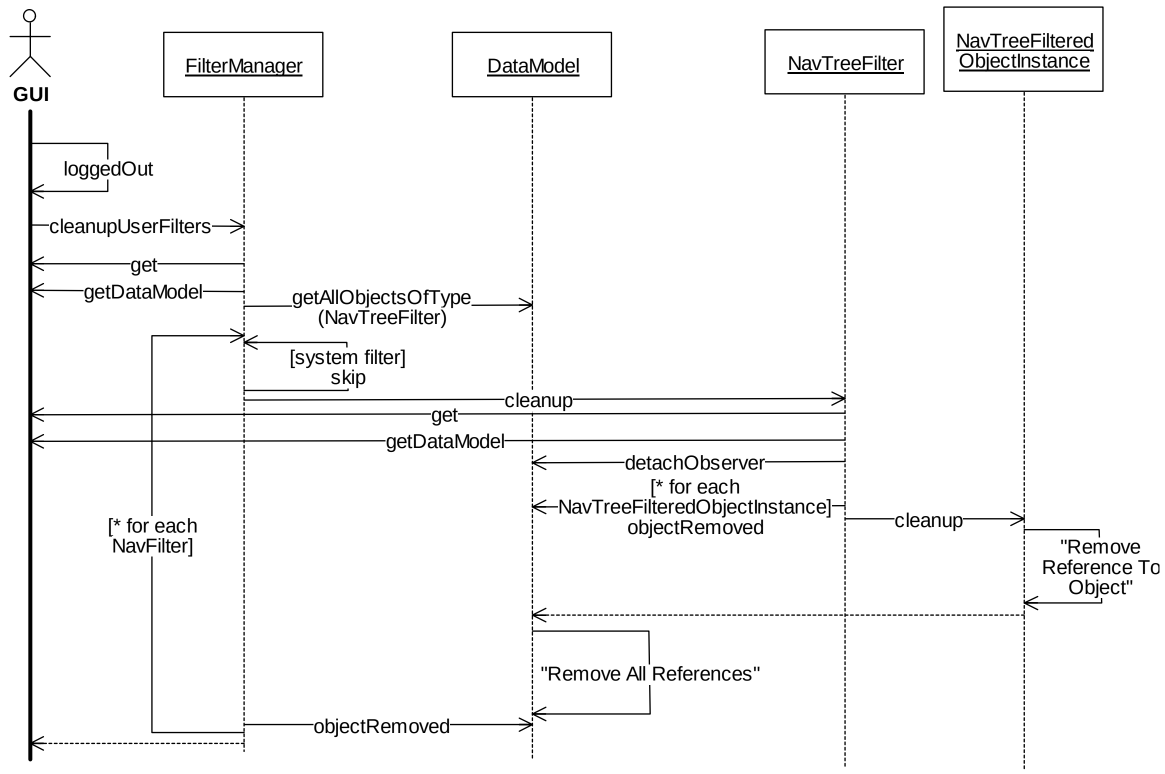


Figure 192. GUIUtility:CleanupUserFilters (Sequence Diagram)

3.14.2.5 GUIUtility:InitializeSystemFilters (Sequence Diagram)

This diagram shows the initialization of the system filters at GUI startup. The GUI calls the FilterManager, which gets the system profiles and attempts to load any existing system filters from the system profile. If successful, it builds the filter hierarchy and puts the filters in the DataModel and attaches the filters to observe the DataModel (see the BuildFilterHierarchy diagram). If no filters are loaded from the system profile, the default system filters have to be created. The FilterManager creates a root filter (the “CHART2” filter) and calls each NavFilterSupporter to create their default system filters. If these filters have a null parent, they are added to the root filter. Then the hierarchy of filters is built and the filters are added to the DataModel and attached to the DataModel as observers (see the BuildFilterHierarchy diagram). Then each filter is saved into the system profile so it can be reconstructed the next time the GUI is restarted.

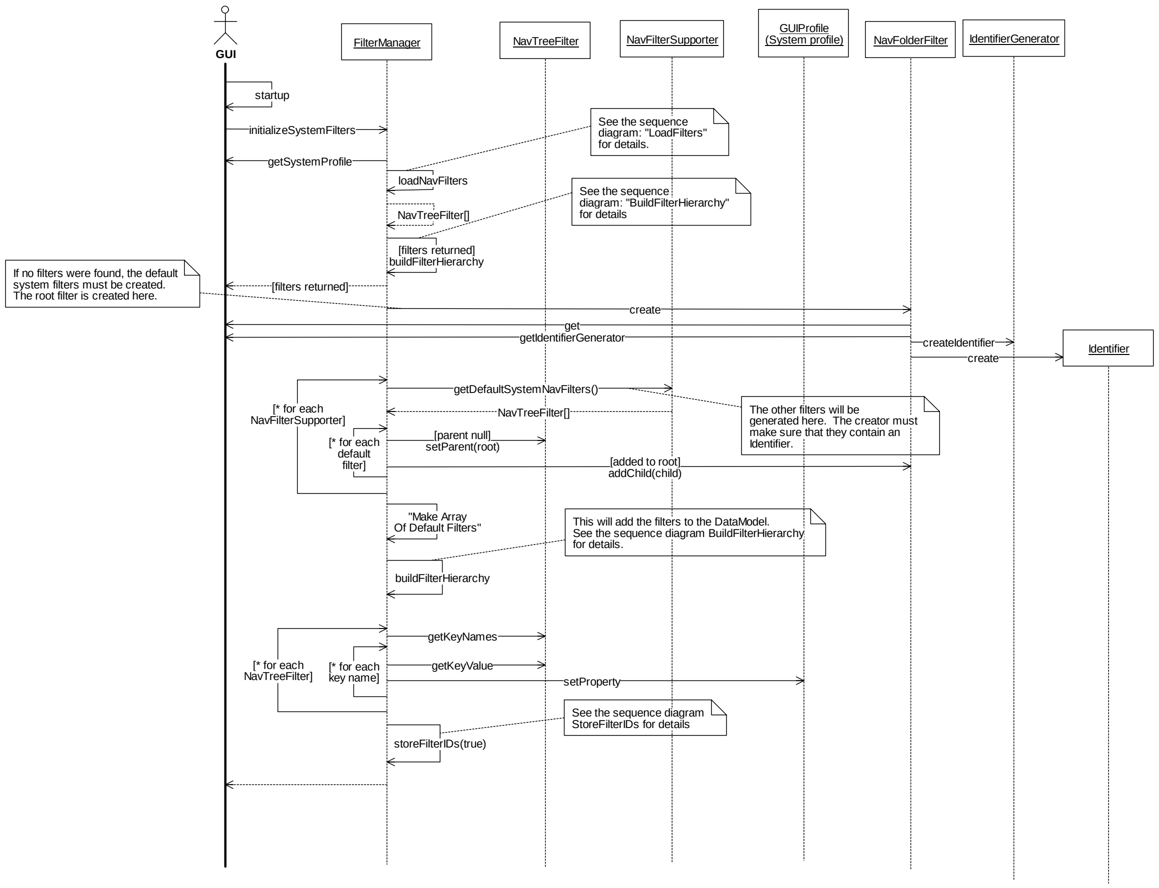

GUI
FilterManager
NavTreeFilter
NavFilterSupporter
GUIProfile (System profile)
NavFolderFilter
IdentifierGenerator
startup
initializeSystemFilters
See the sequence diagram: "LoadFilters" for details.
getSystemProfile
loadNavFilters
See the sequence diagram: "BuildFilterHierarchy" for details
NavTreeFilter[]
[filters returned] buildFilterHierarchy
If no filters were found, the default system filters must be created. The root filter is created here.
[filters returned]
create
get
getIdentifierGenerator
createIdentifier
create
Identifier
getDefaultSystemNavFilters()
[* for each NavFilterSupporter]
NavTreeFilter[]
The other filters will be generated here. The creator must make sure that they contain an Identifier.
[* for each default filter]
[parent null] setParent(root)
[added to root] addChild(child)
"Make Array Of Default Filters"
This will add the filters to the DataModel. See the sequence diagram BuildFilterHierarchy for details.
buildFilterHierarchy
getKeyNames
[* for each NavTreeFilter]
[* for each key name]
getKeyValue
setProperty
See the sequence diagram StoreFilterIDs for details
storeFilterIDs(true)

Figure 193. GUIUtility:InitializeSystemFilters (Sequence Diagram)

3.14.2.6 GUIUtility:InitializeUserFilters (Sequence Diagram)

This diagram shows the creation of the user-specific Navigator filters at login. The GUI calls the FilterManager to load the user filters. The FilterManager then loads the filters (see the LoadFilters diagram) and establishes the filter hierarchy for the new filters, in addition to adding them to the DataModel and attaching them to the DataModel as observers (see the BuildFilterHierarchy diagram).

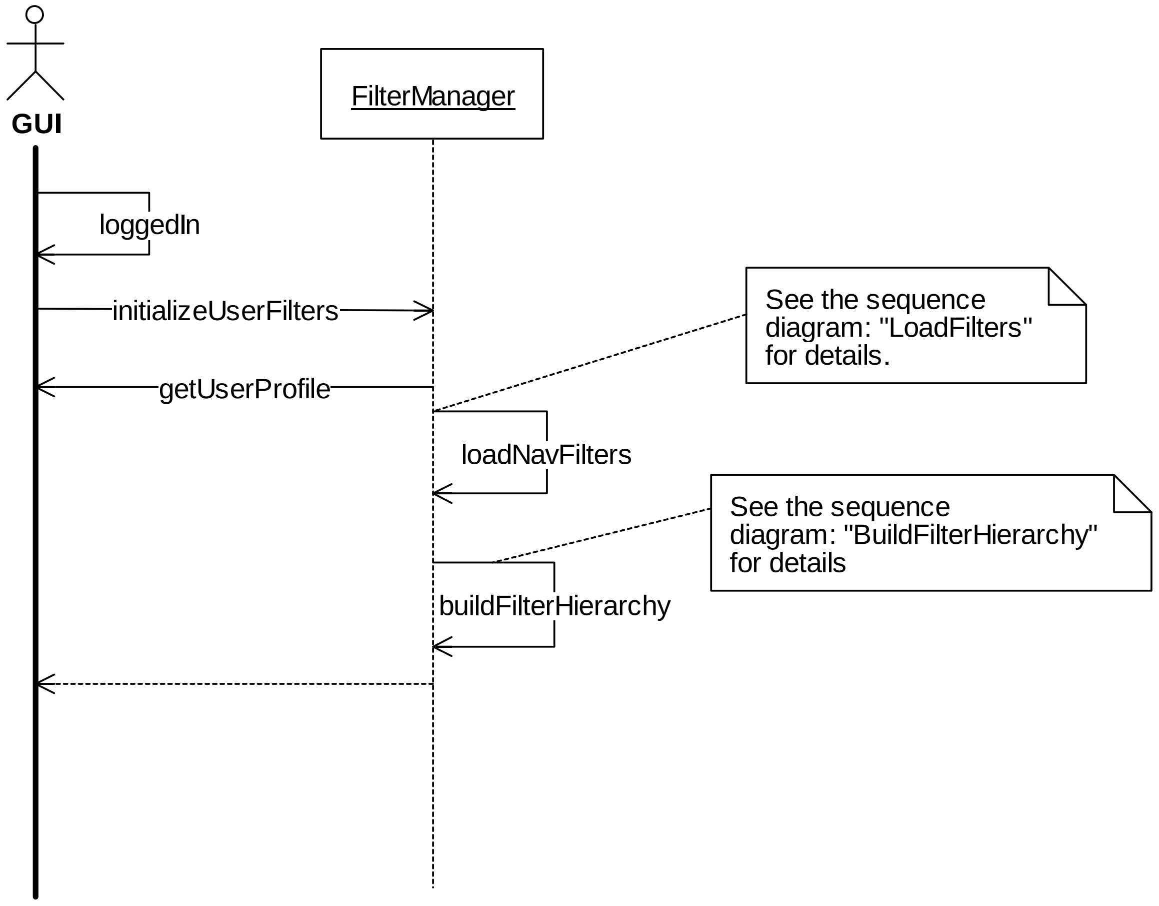

GUI
FilterManager
loggedIn
initializeUserFilters
getUserProfile
loadNavFilters
buildFilterHierarchy
See the sequence diagram: "LoadFilters" for details.
See the sequence diagram: "BuildFilterHierarchy" for details

Figure 194. GUIUtility:InitializeUserFilters (Sequence Diagram)

3.14.2.7 GUIUtility:LoadFilters (Sequence Diagram)

This diagram shows how the filters are loaded, given a GUIProfile object (which can be either a system profile or a user profile). First it queries the filter IDs, which are stored as a delimited sequence of IDs within a given property. Then it separates the IDs. For each filter ID, it gets the class name for the filter and then creates a new instance of that class. Then it asks the new filter object to return the keys (properties) that it supports. Then it queries the profile for the value of each property, and after all of the properties are read, it sets the properties into the filter object.

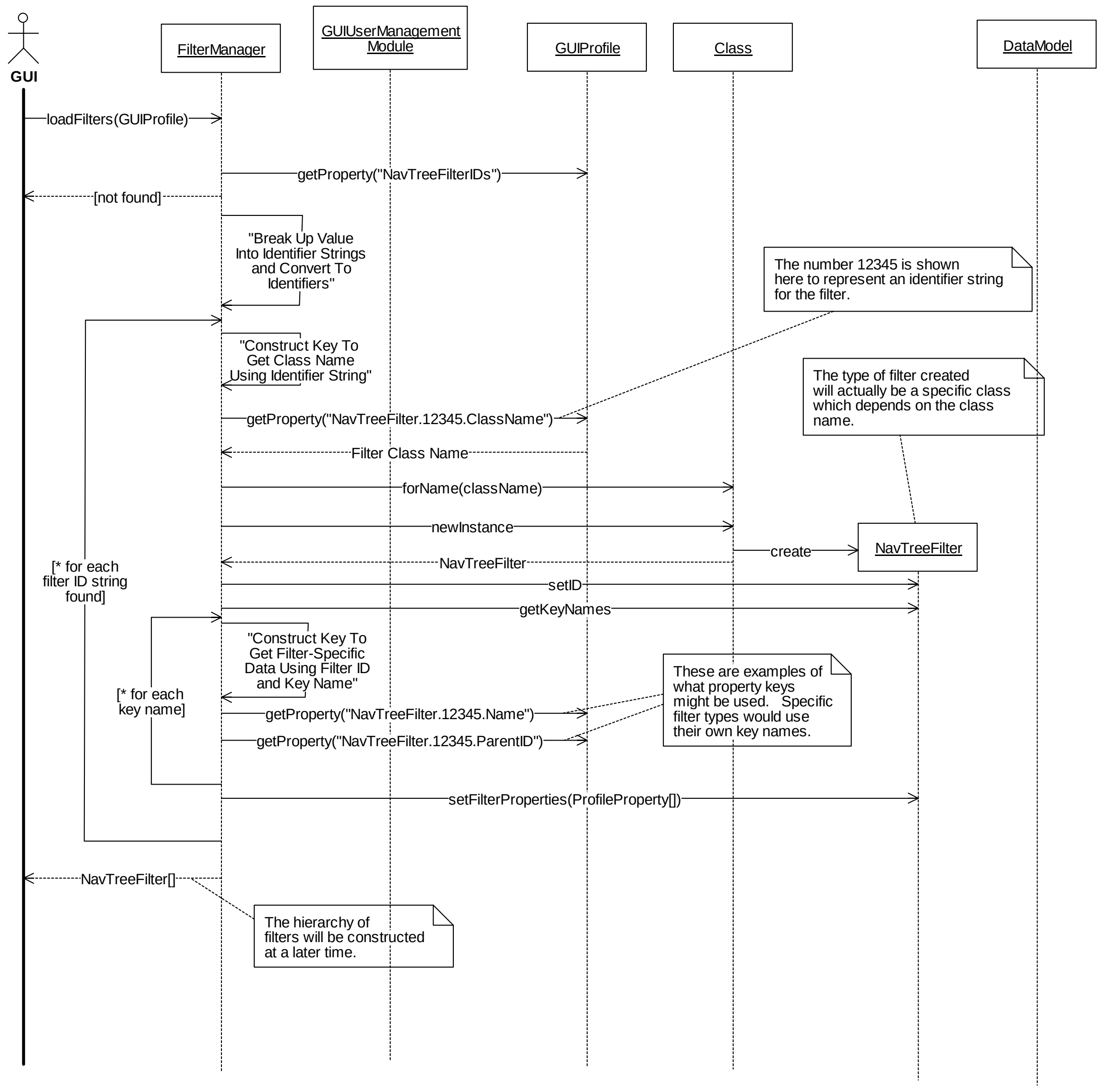

GUI
FilterManager
GUIUserManagement Module
GUIProfile
Class
DataModel
loadFilters(GUIProfile)
getProperty("NavTreeFilterIDs")
[not found]
"Break Up Value Into Identifier Strings and Convert To Identifiers"
The number 12345 is shown here to represent an identifier string for the filter.
"Construct Key To Get Class Name Using Identifier String"
The type of filter created will actually be a specific class which depends on the class name.
getProperty("NavTreeFilter.12345.ClassName")
Filter Class Name
forName(className)
newInstance
create
NavTreeFilter
NavTreeFilter
[* for each filter ID string found]
setID
getKeyNames
"Construct Key To Get Filter-Specific Data Using Filter ID and Key Name"
These are examples of what property keys might be used. Specific filter types would use their own key names.
[* for each key name]
getProperty("NavTreeFilter.12345.Name")
getProperty("NavTreeFilter.12345.ParentID")
setFilterProperties(ProfileProperty[])
NavTreeFilter[]
The hierarchy of filters will be constructed at a later time.

Figure 195. GUIUtility:LoadFilters (Sequence Diagram)

3.14.2.8 GUIUtility:ModifyFilterProperties (Sequence Diagram)

This diagram shows how a filter is modified. The user right clicks on the “Properties” menu item, from the filter’s context menu. The filter calls doProperties() on itself, which invokes the appropriate filter properties dialog corresponding to the specific type of filter. When the user clicks “OK”, the dialog will set the properties into the filter. The filter will then get the GUIProfile object (either the system profile or user profile, depending on which type of filter it is) and will save the properties into the GUIProfile. (See the sequence diagram UpdateForFilterChange for more details).

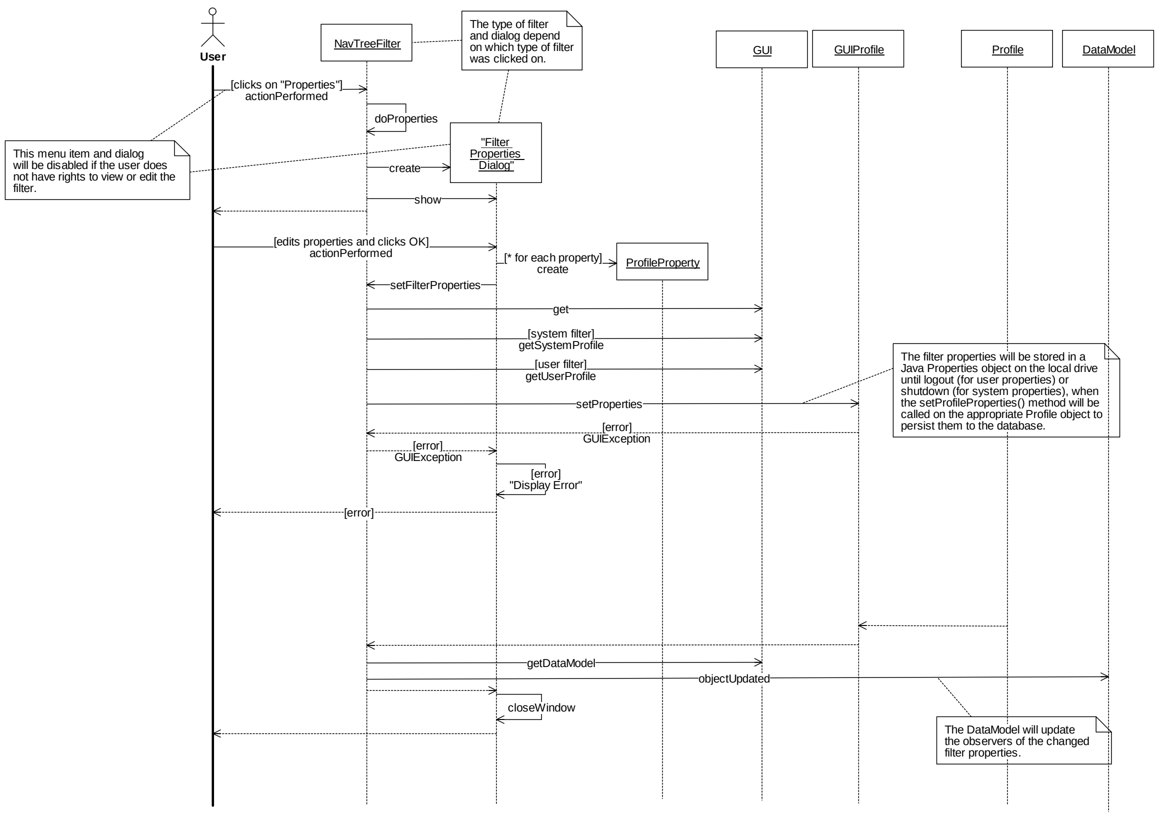
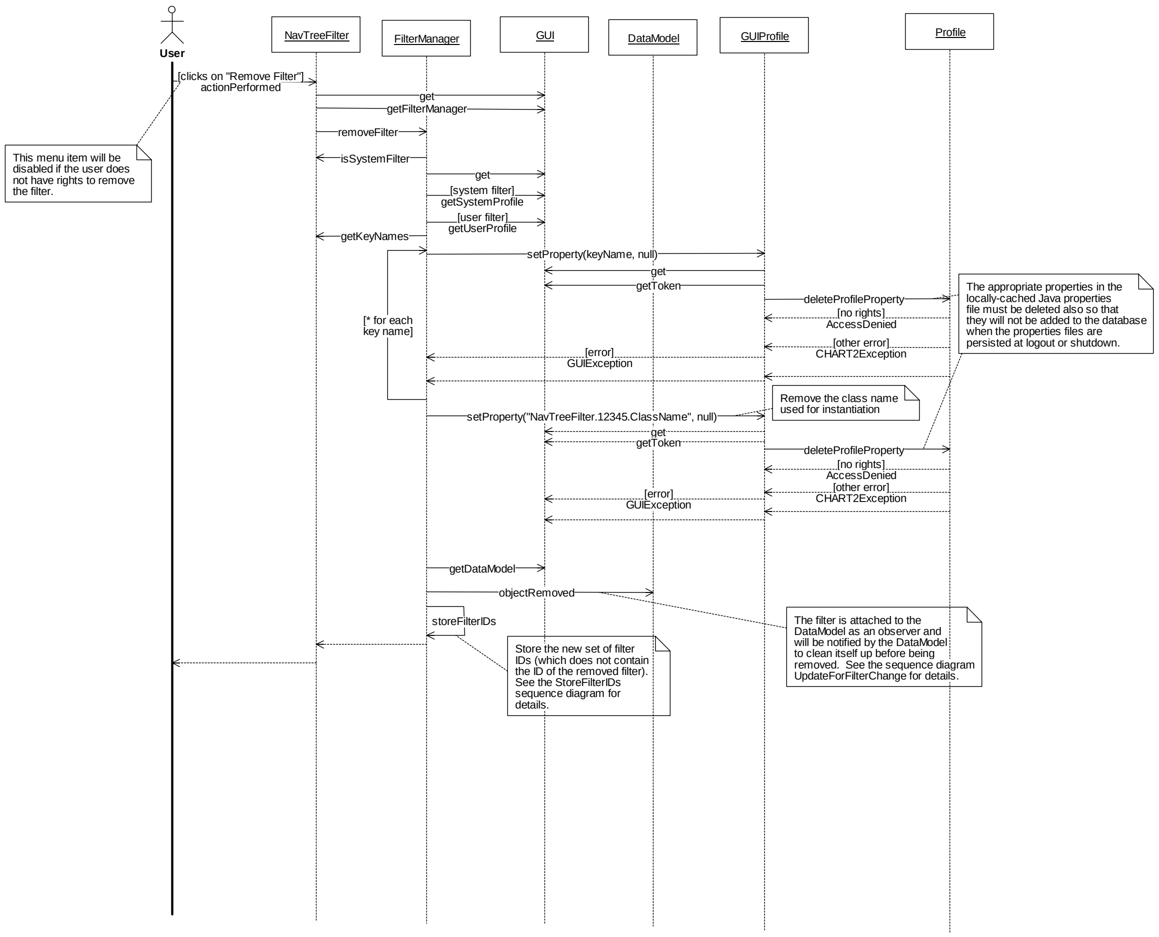

User
NavTreeFilter
The type of filter and dialog depend on which type of filter was clicked on.
GUI
GUIProfile
Profile
DataModel
[clicks on "Properties"] actionPerformed
doProperties
This menu item and dialog will be disabled if the user does not have rights to view or edit the filter.
"Filter Properties Dialog"
create
show
[edits properties and clicks OK] actionPerformed
[* for each property] create
ProfileProperty
setFilterProperties
get
[system filter] getSystemProfile
[user filter] getUserProfile
The filter properties will be stored in a Java Properties object on the local drive until logout (for user properties) or shutdown (for system properties), when the setProfileProperties() method will be called on the appropriate Profile object to persist them to the database.
setProperties
[error] GUIException
[error] GUIException
[error] "Display Error"
[error]
getDataModel
objectUpdated
closeWindow
The DataModel will update the observers of the changed filter properties.

Figure 196. GUIUtility:ModifyFilterProperties (Sequence Diagram)

3.14.2.9 GUIUtility:RemoveFilter (Sequence Diagram)

This diagram shows how a filter is removed from the system. The user clicks on the "Remove Filter" menu item from the filter's context menu item. The FilterManager then gets the appropriate GUIProfile (either system or user), and after asking the filter for all of the key names which it supports, sets all of the property values to null in the GUIProfile. The GUIProfile will then delete the properties. Finally, the classname will be deleted in the profile and the filter IDs will be stored (after removing the filter from the DataModel). (See the UpdateForFilterChange sequence diagram for more details about the cleanup of the filter).



User
NavTreeFilter
FilterManager
GUI
DataModel
GUIProfile
Profile
[clicks on "Remove Filter"] actionPerformed
get
getFilterManager
removeFilter
isSystemFilter
This menu item will be disabled if the user does not have rights to remove the filter.
get
[system filter] getSystemProfile
[user filter] getUserProfile
getKeyNames
setProperty(keyName, null)
get
getToken
[* for each key name]
deleteProfileProperty
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
The appropriate properties in the locally-cached Java properties file must be deleted also so that they will not be added to the database when the properties files are persisted at logout or shutdown.
Remove the class name used for instantiation
setProperty("NavTreeFilter.12345.ClassName", null)
get
getToken
deleteProfileProperty
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
getDataModel
objectRemoved
storeFilterIDs
The filter is attached to the DataModel as an observer and will be notified by the DataModel to clean itself up before being removed. See the sequence diagram UpdateForFilterChange for details.
Store the new set of filter IDs (which does not contain the ID of the removed filter). See the StoreFilterIDs sequence diagram for details.

Figure 197. GUIUtility:RemoveFilter (Sequence Diagram)

3.14.2.10 GUIUtility:StoreFilterIDs (Sequence Diagram)

This diagram shows how the filter IDs are stored in a GUIProfile. Storing the IDs enables the filters to be reconstructed at startup (for system filters) or login (for user filters). The FilterManager gets all of the NavTreeFilter objects from the DataModel, and appends the string IDs of those filters which have the same system/user flag as was requested. The FilterManager then gets the appropriate GUIProfile and saves the concatenated value into it.

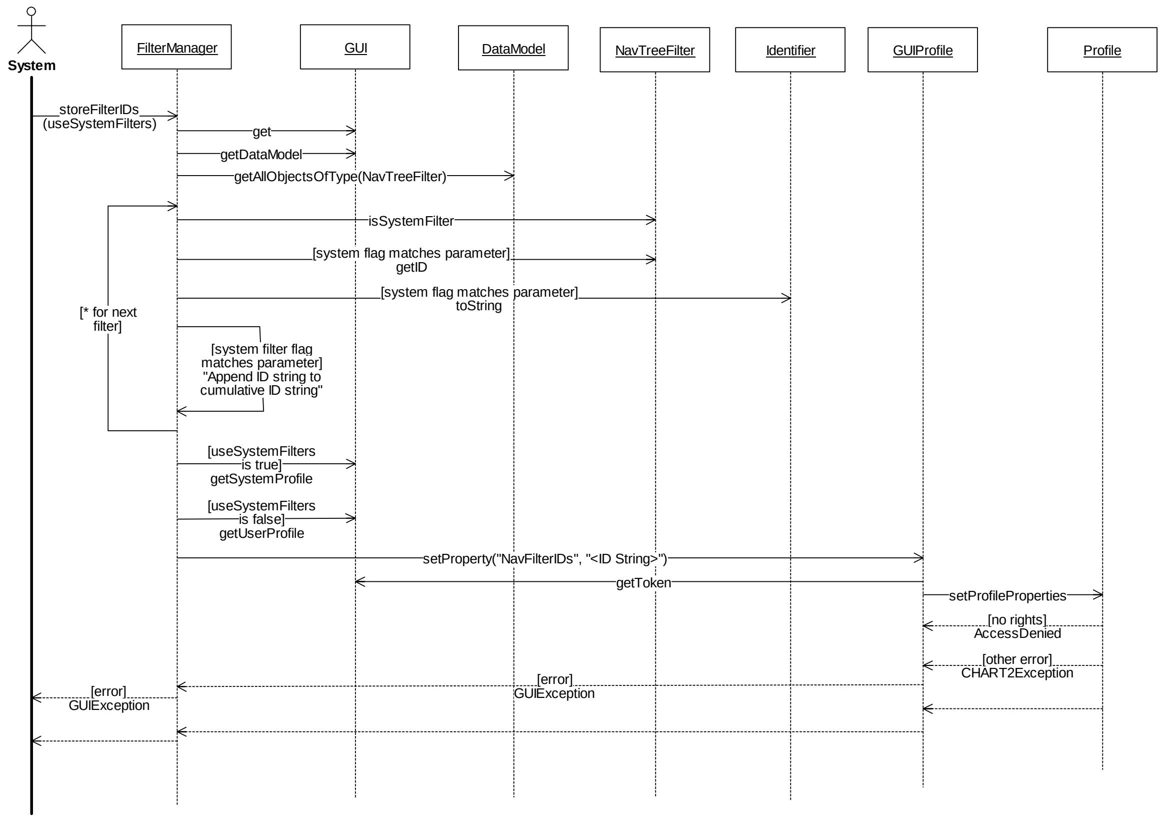

System
FilterManager
GUI
DataModel
NavTreeFilter
Identifier
GUIProfile
Profile
storeFilterIDs (useSystemFilters)
get
getDataModel
getAllObjectsOfType(NavTreeFilter)
isSystemFilter
[system flag matches parameter] getID
[system flag matches parameter] toString
[* for next filter]
[system filter flag matches parameter] "Append ID string to cumulative ID string"
[useSystemFilters is true] getSystemProfile
[useSystemFilters is false] getUserProfile
setProperty("NavFilterIDs", "<ID String>")
getToken
setProfileProperties
[no rights] AccessDenied
[other error] CHART2Exception
[error] GUIException
[error] GUIException

Figure 198. GUIUtility:StoreFilterIDs (Sequence Diagram)

3.14.2.11 GUIUtility:UpdateForFilterChange (Sequence Diagram)

This diagram shows the processing that occurs when a filter has been added, modified, or removed from the DataModel. The filter is an observer of the DataModel, so it also catches the updates for any changes to itself. If the filter was added or changed, the filter gets all of the NavTreeDisplayable objects from the DataModel and filters them. The root filter is called first, then each ancestor down to this filter. Any NavTreeObjectInstances that were contained in the filter, but whose objects they represent are not in the newly filtered set, are removed. Any NavTreeDisplayable objects in the newly filtered set but not currently contained in the filter are wrapped with NavTreeFilteredObjectInstance objects, which then are added to the DataModel and the filter. If the filter was removed from the DataModel, then all of the NavTreeFilteredObjectInstance objects are removed from the filter and the DataModel.

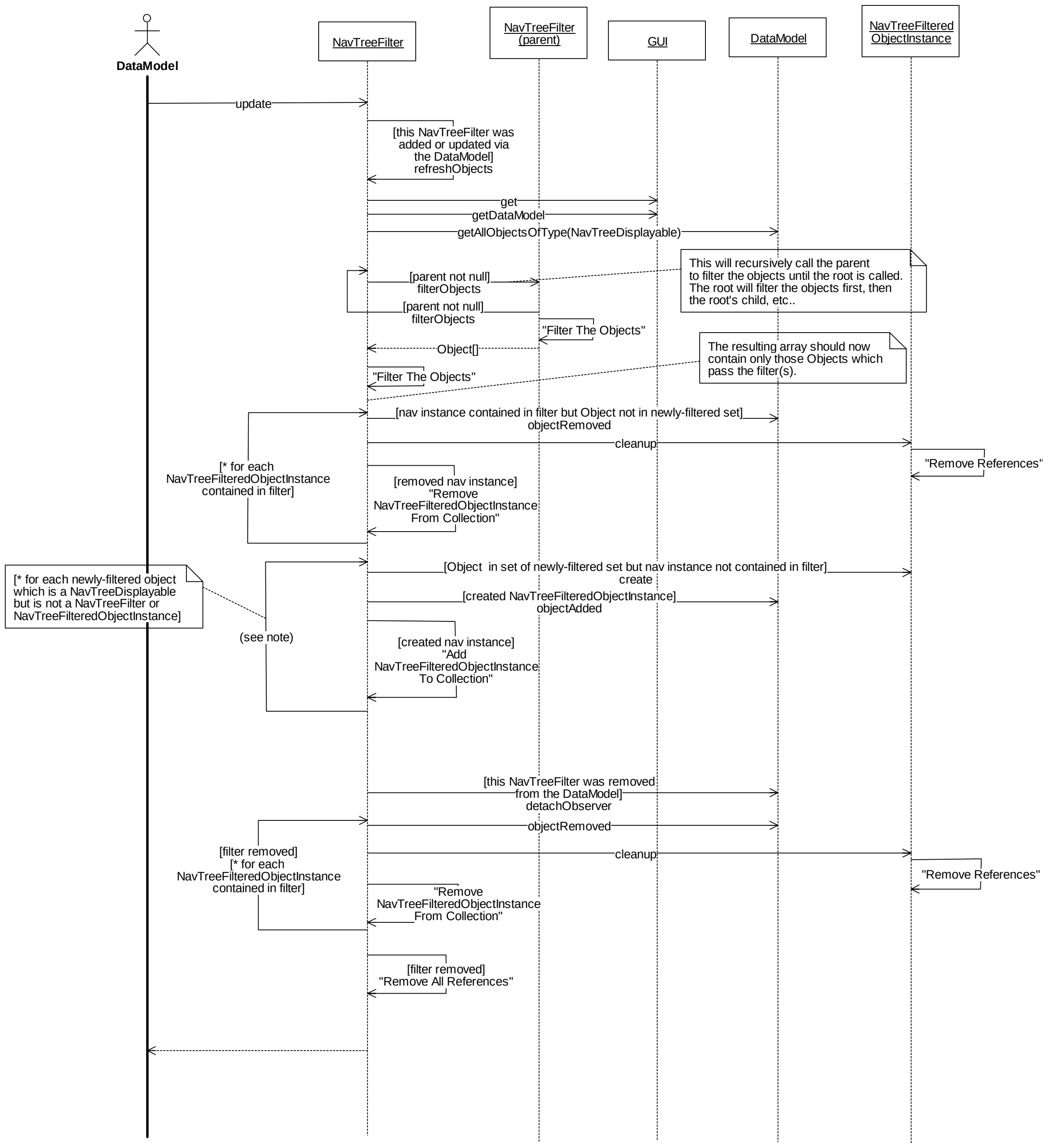

DataModel
NavTreeFilter
NavTreeFilter (parent)
GUI
DataModel
NavTreeFiltered ObjectInstance
update
[this NavTreeFilter was added or updated via the DataModel] refreshObjects
get
getDataModel
getAllObjectsOfType(NavTreeDisplayable)
[parent not null] filterObjects
[parent not null] filterObjects
This will recursively call the parent to filter the objects until the root is called. The root will filter the objects first, then the root's child, etc..
"Filter The Objects"
Object[]
The resulting array should now contain only those Objects which pass the filter(s).
"Filter The Objects"
[nav instance contained in filter but Object not in newly-filtered set] objectRemoved
cleanup
"Remove References"
[* for each NavTreeFilteredObjectInstance contained in filter]
[removed nav instance] "Remove NavTreeFilteredObjectInstance From Collection"
[* for each newly-filtered object which is a NavTreeDisplayable but is not a NavTreeFilter or NavTreeFilteredObjectInstance]
[Object in set of newly-filtered set but nav instance not contained in filter] create
[created NavTreeFilteredObjectInstance] objectAdded
(see note)
[created nav instance] "Add NavTreeFilteredObjectInstance To Collection"
[this NavTreeFilter was removed from the DataModel] detachObserver
objectRemoved
cleanup
[filter removed] [* for each NavTreeFilteredObjectInstance contained in filter]
"Remove References"
"Remove NavTreeFilteredObjectInstance From Collection"
[filter removed] "Remove All References"

Figure 199. GUIUtility:UpdateForFilterChange (Sequence Diagram)

3.14.2.12 GUIUtility:UpdateForObjectChanges (Sequence Diagram)

This diagram shows the processing that occurs when objects have been added, modified, or removed from the DataModel. The filter is an observer of the DataModel, so it catches the updates for any changes to objects. It takes all of the objects that were added or changed and filters them. The root filter is called first, then each ancestor down to this filter. Any NavTreeObjectInstances that were contained in the filter, but whose objects they represent are not in the newly-filtered set, are removed. Any NavTreeDisplayable objects in the newly-filtered set but not currently contained in the filter are wrapped with NavTreeFilteredObjectInstance objects, which then are added to the DataModel and the filter. If the filter was removed from the DataModel, then all of the NavTreeFilteredObjectInstance objects are removed from the filter and the DataModel. For any objects that were removed from the DataModel, if they have NavTreeFilteredObjectInstance objects wrapping them, these wrappers are removed from the DataModel and from the filter.

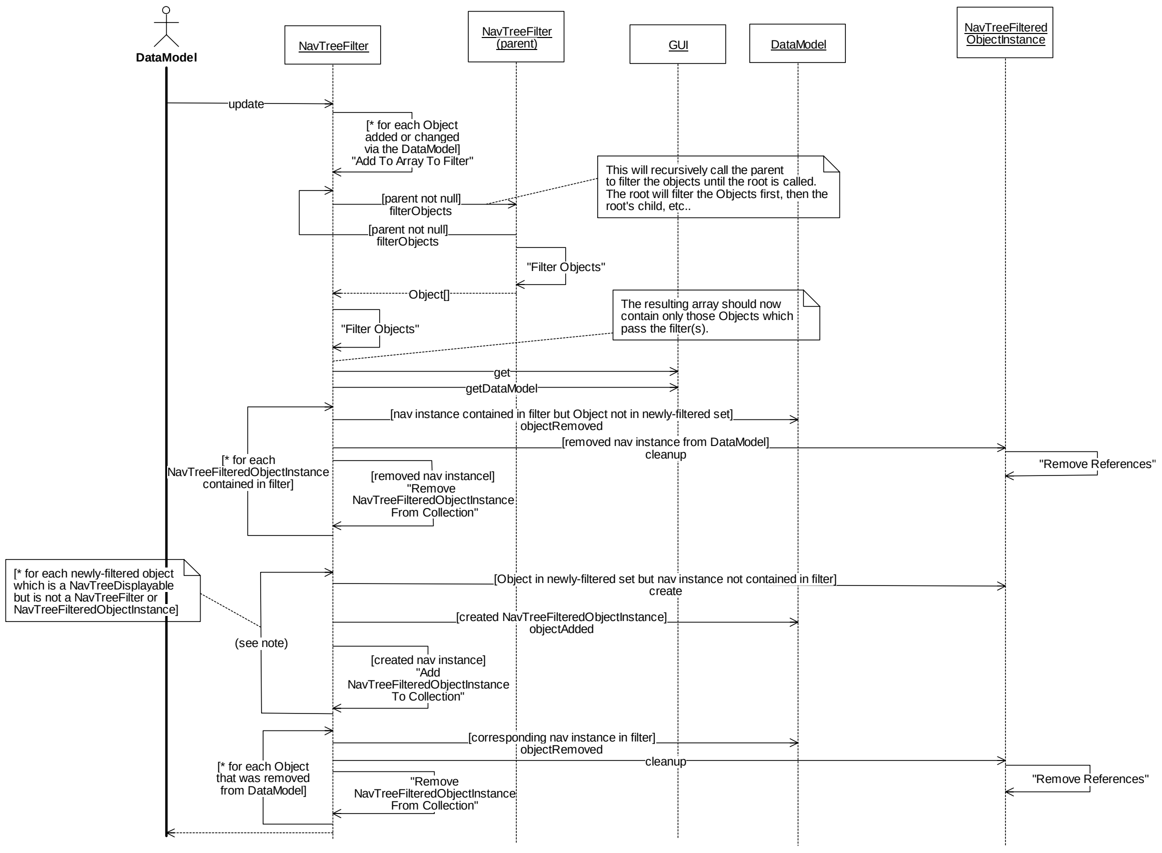

DataModel
NavTreeFilter
NavTreeFilter (parent)
GUI
DataModel
NavTreeFiltered ObjectInstance
update
[* for each Object added or changed via the DataModel] "Add To Array To Filter"
This will recursively call the parent to filter the objects until the root is called. The root will filter the Objects first, then the root's child, etc..
[parent not null] filterObjects
[parent not null] filterObjects
"Filter Objects"
Object[]
The resulting array should now contain only those Objects which pass the filter(s).
"Filter Objects"
get
getDataModel
[nav instance contained in filter but Object not in newly-filtered set] objectRemoved
[removed nav instance from DataModel] cleanup
"Remove References"
[* for each NavTreeFilteredObjectInstance contained in filter]
[removed nav instance] "Remove NavTreeFilteredObjectInstance From Collection"
[* for each newly-filtered object which is a NavTreeDisplayable but is not a NavTreeFilter or NavTreeFilteredObjectInstance]
[Object in newly-filtered set but nav instance not contained in filter] create
[created NavTreeFilteredObjectInstance] objectAdded
(see note)
[created nav instance] "Add NavTreeFilteredObjectInstance To Collection"
[corresponding nav instance in filter] objectRemoved
cleanup
"Remove References"
[* for each Object that was removed from DataModel]
"Remove NavTreeFilteredObjectInstance From Collection"

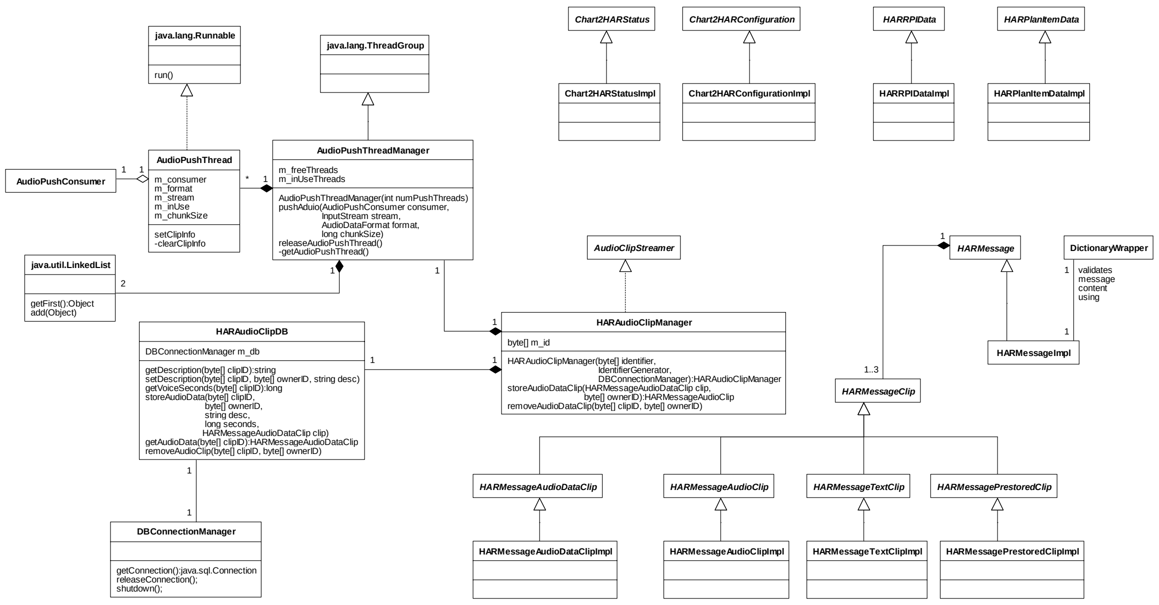
Figure 200. GUIUtility:UpdateForObjectChanges (Sequence Diagram)

3.15 HARUtility

3.15.1 Class Diagrams

3.15.1.1 HARUtility (Class Diagram)

This class diagram shows classes related to the HAR that are used by both the GUI and the server. Most (if not all) of these classes are implementations of value type classes defined in the system interfaces (IDL).



java.lang.Runnable
run()
java.lang.ThreadGroup
Chart2HARStatus
Chart2HARConfiguration
HARRPIData
HARPlanItemData
Chart2HARStatusImpl
Chart2HARConfigurationImpl
HARRPIDataImpl
HARPlanItemDataImpl
AudioPushConsumer
1
1
AudioPushThread
m_consumer
m_format
m_stream
m_inUse
m_chunkSize
setClipInfo
-clearClipInfo
*
1
AudioPushThreadManager
m_freeThreads
m_inUseThreads
AudioPushThreadManager(int numPushThreads)
pushAduio(AudioPushConsumer consumer, InputStream stream, AudioDataFormat format, long chunkSize)
releaseAudioPushThread()
-getAudioPushThread()
java.util.LinkedList
getFirst():Object
add(Object)
2
1
1
AudioClipStreamer
HARMessage
DictionaryWrapper
1
validates message content using
1
HARMessageImpl
HARAudioClipDB
DBConnectionManager m_db
getDescription(byte[] clipID):string
setDescription(byte[] clipID, byte[] ownerID, string desc)
getVoiceSeconds(byte[] clipID):long
storeAudioData(byte[] clipID, byte[] ownerID, string desc, long seconds, HARMessageAudioDataClip clip)
getAudioData(byte[] clipID):HARMessageAudioDataClip
removeAudioClip(byte[] clipID, byte[] ownerID)
1
1
1
1
HARAudioClipManager
byte[] m_id
HARAudioClipManager(byte[] identifier, IdentifierGenerator, DBConnectionManager):HARAudioClipManager
storeAudioDataClip(HARMessageAudioDataClip clip, byte[] ownerID):HARMessageAudioClip
removeAudioDataClip(byte[] clipID, byte[] ownerID)
1..3
HARMessageClip
1
1
HARMessageAudioDataClip
HARMessageAudioClip
HARMessageTextClip
HARMessagePrestoredClip
DBConnectionManager
getConnection():java.sql.Connection
releaseConnection();
shutdown();
HARMessageAudioDataClipImpl
HARMessageAudioClipImpl
HARMessageTextClipImpl
HARMessagePrestoredClipImpl

Figure 201. HARUtility (Class Diagram)

3.15.1.1.1 AudioClipStreamer (Class)

This interface is implemented by objects that can push a previously stored audio clip given its ID. The audio data is pushed via the AudioPushConsumer supplied by the user of this interface

3.15.1.1.2 AudioPushConsumer (Class)

This interface is implemented by objects that wish to receive audio data using the push model, where the server pushes the data to the consumer. One call to pushAudioProperties() will always precede any calls to pushAudio().

3.15.1.1.3 AudioPushThread (Class)

This class is a thread that is used to push audio clip information to an AudioPushConsumer.

3.15.1.1.4 AudioPushThreadManager (Class)

This class maintains a pool of reusable AudioPushThread objects, which can be used to push audio clip information back to the client. It provides the functionality to manage access to the AudioPushThreads.

3.15.1.1.5 Chart2HARConfiguration (Class)

This class contains configuration data for the HAR that is used for CHART II specific processing (as opposed to the configuration values contained in HARConfiguration that relate to typical HAR usage).

3.15.1.1.6 Chart2HARConfigurationImpl (Class)

This class is a concrete implementation of the Chart2HARConfiguration abstract class generated from IDL.

3.15.1.1.7 Chart2HARStatus (Class)

This class contains status information for a Chart2HAR object. This information is specific to Chart II processing and extends beyond the status related to typical HAR device control.

3.15.1.1.8 Chart2HARStatusImpl (Class)

This class is a concrete implementation of the Chart2HARStatus abstract class generated from IDL.

3.15.1.1.9 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the “jdbc.drivers” system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.15.1.1.10 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerance by automatically failing over to a “working” dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.15.1.1.11 HARAudioClipDB (Class)

This class provides access to the database for the HARAudioClipManager. It provides a means to store and retrieve recorded voice to/from the database.

3.15.1.1.12 HARAudioClipManager (Class)

This class provides the implementation of the AudioStreamer interface and is capable of streaming recorded audio clips that have been previously stored. When requested to stream an audio clip, this class pulls the audio data from its persistent store pushes the audio data to the given AudioPushConsumer in a worker thread. This class also allows newly recorded audio clips to be added to the system. When a clip is added to the system it is assigned a unique ID and a HARMessageAudioClip is created as a thin wrapper to provide access to the audio data. When new audio clips are added to the system, the ID of the owner is passed to facilitate clean-up of the clip when it is no longer needed.

3.15.1.1.13 HARMessage (Class)

This utility class represents a message that is capable of being stored on a HAR. It stores the HAR message as a HAR message header, body and footer. It contains methods to input and output them in different formats.

3.15.1.1.14 HARMessageAudioClip (Class)

This class is a thin wrapper for recorded voice that is to be played on a HAR. This class is passed around the system instead of passing the actual voice data. When the actual voice data is needed to play to the user or to program the HAR device, this object’s streamer is used to stream the actual voice data.

3.15.1.1.15 HARMessageAudioClipImpl (Class)

This class defines HARMessageAudioClip as defined in the IDL. Refer to HARMessageAudioClip for details.

3.15.1.1.16 HARMessageAudioDataClip (Class)

This class is a message clip that contains audio data and the format of the audio data. Because audio data can be very large, this type of clip is reserved for use when recorded voice is first entered into the system. Recorded voice that already exists in the system is passed throughout the system using HARMessageAudioClip to avoid sending the large audio data when possible.

3.15.1.1.17 HARMessageAudioDataClipImpl (Class)

This class implements the HARMessageAudioDataClip as defined in the IDL. Refer to HARMessageAudioDataClip for details.

3.15.1.1.18 HARMessageClip (Class)

This class represents a section of a HAR message. It can be either plain text that would need to be converted to audio prior to broadcast, or binary format (MP3, WAV, etc.)

3.15.1.1.19 HARMessageImpl (Class)

This class is a concrete implementation of the HARMessage abstract class generated from IDL.

3.15.1.1.20 HARMessagePrestoredClip (Class)

This class stores data used to identify the usage of a clip that has already been stored on a HAR device.

3.15.1.1.21 HARMessagePrestoredClipImpl (Class)

This class implements HARMessagePrestoredClip as defined in IDL. Refer to HARMessagePrestoredClip for details.

3.15.1.1.22 HARMessageTextClip (Class)

This class represents a HAR message content object that is in plain text format. This message can be checked for banned words and will be converted into a voice message using a speech engine to broadcast on a HAR device.

3.15.1.1.23 HARMessageTextClipImpl (Class)

This class implements HARMessageTextClip as defined in the IDL. Refer to HARMessageTextClip for details.

3.15.1.1.24 HARPlanItemData (Class)

This class is used to associate a message with a HAR for use in Plans.

3.15.1.1.25 HARPlanItemDataImpl (Class)

This class is a concrete implementation of the HARPlanItemData abstract class generated from IDL.

3.15.1.1.26 HARRPIData (Class)

This class represents an item in a traffic event response plan that is capable of issuing a command to put a message on a HAR when executed. When the item is executed, it adds the message to the arbitration queue of the specified HAR. When the item is removed from the response plan (manually or implicitly through closing the traffic event) the item asks the HAR's arbitration queue to remove the message.

3.15.1.1.27 HARRPIDataImpl (Class)

This class is a concrete implementation of the HARRPIData abstract class generated from IDL.

3.15.1.1.28 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.15.1.1.29 java.lang.ThreadGroup (Class)

A thread group represents a set of threads.

3.15.1.1.30 java.util.LinkedList (Class)

This class is an implementation of List interface for a linked list.

3.15.2 Sequence Diagrams

3.15.2.1 HARUtility:PushAudio (Sequence Diagram)

This diagram shows how audio data is pushed back to the client. The AudioPushThreadManager manages a pool of threads that can be used to push audio data back to the clients. When a request is made to push audio, the AudioPushThreadManager looks in the thread list for a free thread. If all the threads are being used, the request waits until a thread becomes available. Once a thread becomes available, the thread is notified of the clip by setting the clip data and the thread starts pushing the audio data by first pushing the audio properties. Then, the thread starts to push the audio data in chunks of the size requested by the client. If the pushing operation fails, an error is passed to the consumer. At the completion of pushing, the thread clears the clip data and informs the AudioPushThreadManager to free the thread. The AudioPushThreadManager in turn frees the thread and notifies any waiting request.

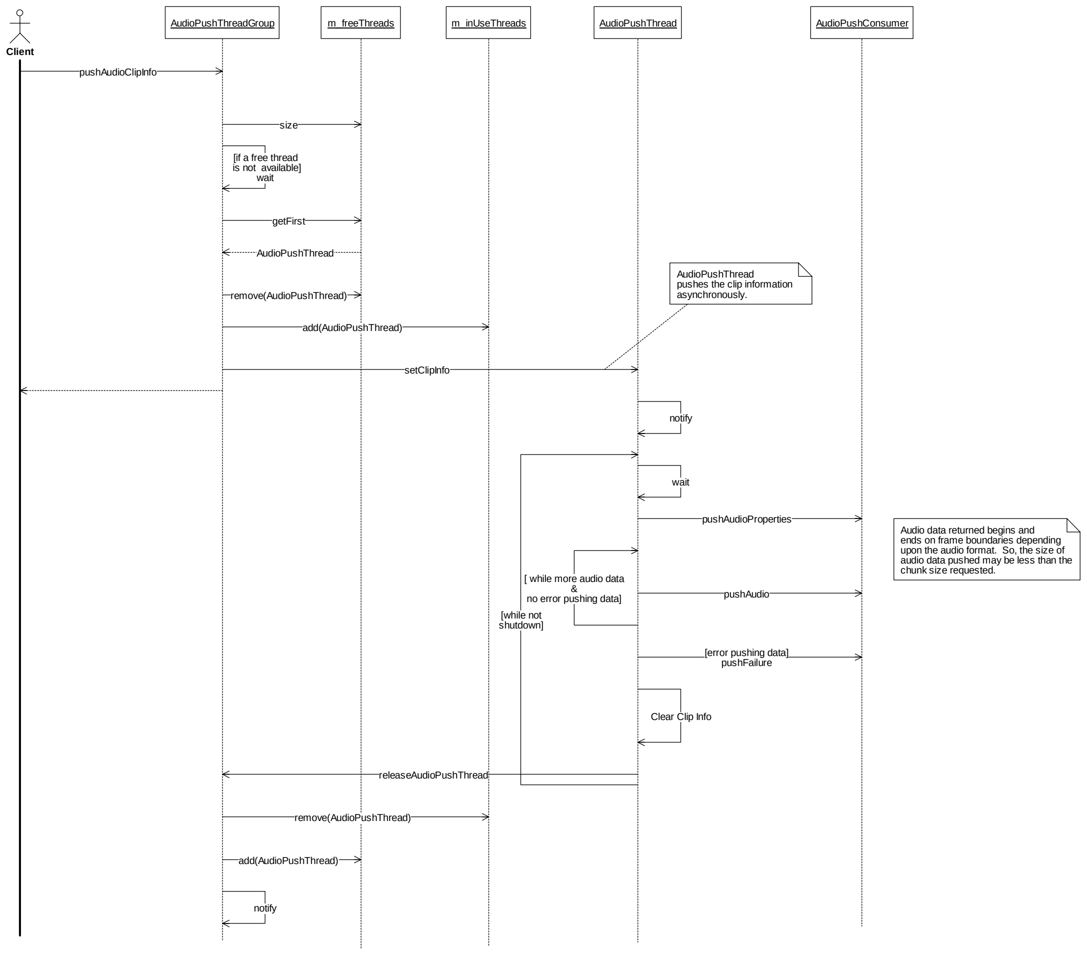

Client
AudioPushThreadGroup
m_freeThreads
m_inUseThreads
AudioPushThread
AudioPushConsumer
pushAudioClipInfo
size
[if a free thread is not available] wait
getFirst
AudioPushThread
remove(AudioPushThread)
add(AudioPushThread)
AudioPushThread pushes the clip information asynchronously.
setClipInfo
notify
wait
pushAudioProperties
Audio data returned begins and ends on frame boundaries depending upon the audio format. So, the size of audio data pushed may be less than the chunk size requested.
[while more audio data & no error pushing data]
pushAudio
[while not shutdown]
[error pushing data] pushFailure
Clear Clip Info
releaseAudioPushThread
remove(AudioPushThread)
add(AudioPushThread)
notify

Figure 202. HARUtility:PushAudio (Sequence Diagram)

3.15.2.2 HARUtility:StoreAudioClip (Sequence Diagram)

When a Chart2HARImpl or the MessageLibraryDB object have been passed a HAR message that contains a HARMessageAudioDataClip, the HARAudioClipManager is called to store the voice data and create a thin wrapper object that represents the voice data. This thin wrapper is passed around the system instead of the voice data itself. The thin wrapper contains a reference to the HARAudioClipManager which will push the voice data to any holders of the thin wrapper that request the actual voice data.

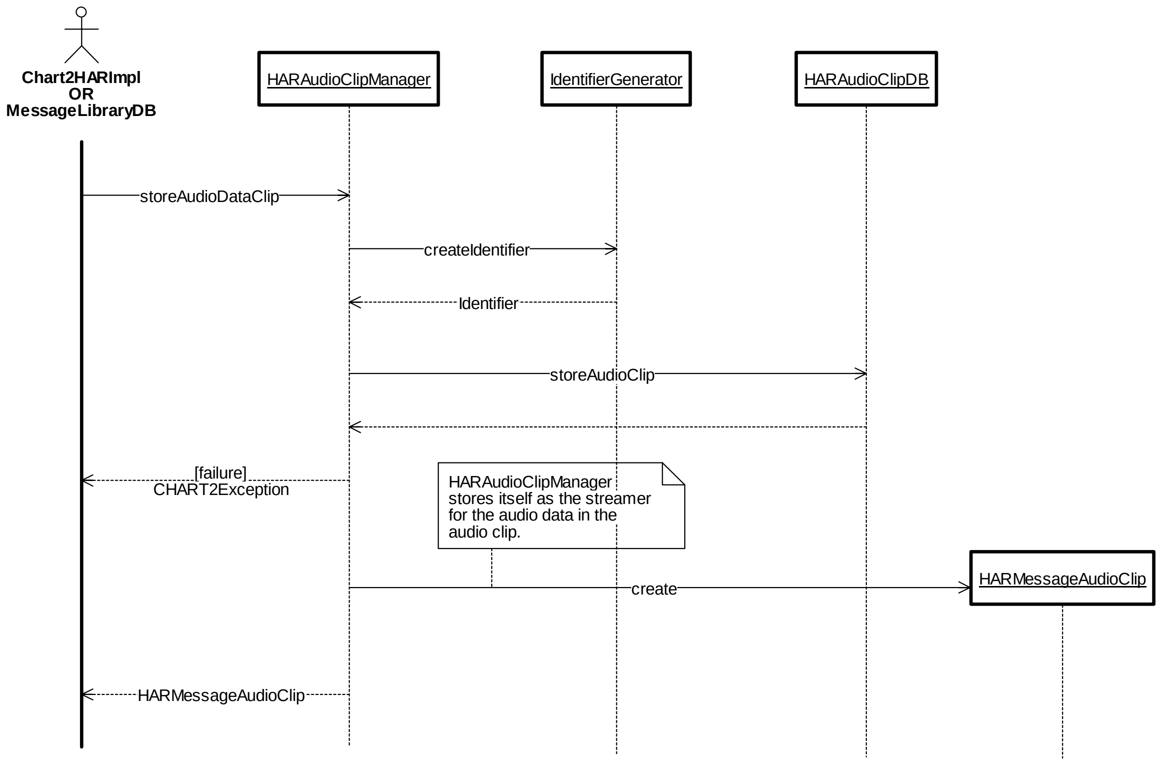


Figure 203. HARUtility:StoreAudioClip (Sequence Diagram)

3.16 Java Classes

3.16.1 Class Diagrams

3.16.1.1 JavaClasses (Class Diagram)

This package is included for reference to classes included in the Java programming language that are used in class and sequence diagrams for other packages within this design.

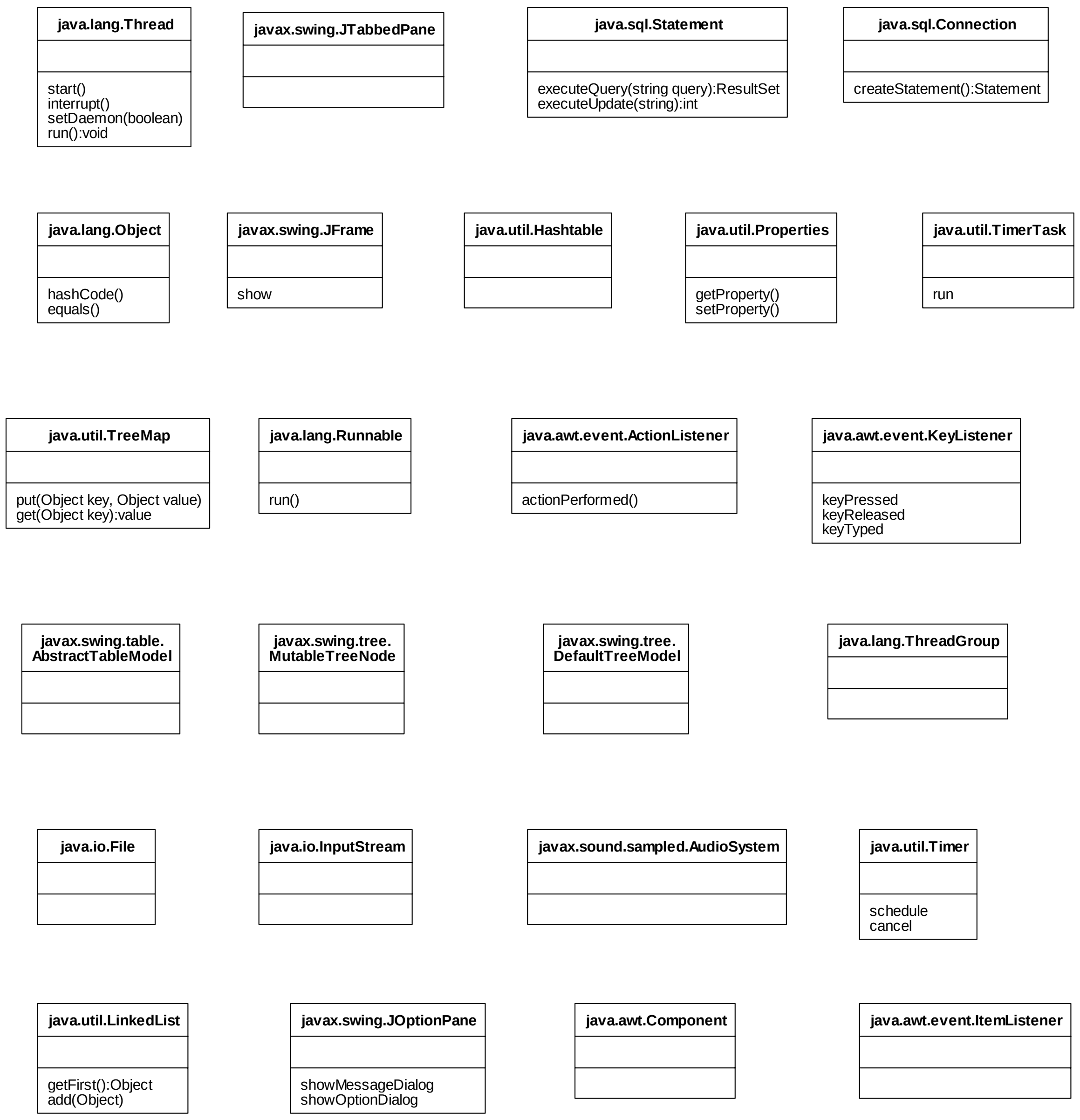


Figure 204. JavaClasses (Class Diagram)

3.16.1.1.1 java.awt.Component (Class)

This class is the base class for all graphical user interface components such as buttons and panels.

3.16.1.1.2 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.16.1.1.3 java.awt.event.ItemListener (Class)

This interface allows the implementing class to listen for changes to an item such as a list item or combo box item.

3.16.1.1.4 java.awt.event.KeyListener (Class)

Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

3.16.1.1.5 java.io.File (Class)

This class is an abstract representation of file and directory pathnames.

3.16.1.1.6 java.io.InputStream (Class)

Java class that represents a input stream of bytes.

3.16.1.1.7 java.lang.Object (Class)

This is the base class from which all Java classes inherit.

3.16.1.1.8 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java’s threading mechanism.

3.16.1.1.9 java.lang.Thread (Class)

This class represents a java thread of execution.

3.16.1.1.10 java.lang.ThreadGroup (Class)

A thread group represents a set of threads.

3.16.1.1.11 java.sql.Connection (Class)

This class represents a connection (session) with a specific database.

3.16.1.1.12 java.sql.Statement (Class)

Java class used for executing a static SQL statement and obtaining the results produced by it.

3.16.1.1.13 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

3.16.1.1.14 java.util.LinkedList (Class)

This class is an implementation of List interface for a linked list.

3.16.1.1.15 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its “defaults”; this second property list is searched if the property key is not found in the original property list.

3.16.1.1.16 java.util.Timer (Class)

This class provides asynchronous execution of tasks that are scheduled for one-time or recurring execution.

3.16.1.1.17 java.util.TimerTask (Class)

This class is an abstract base class which can be scheduled with a timer to be executed one or more times.

3.16.1.1.18 java.util.TreeMap (Class)

This class is an implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key’s class, or by the comparator provided at creation time, depending on which constructor is used.

3.16.1.1.19 javax.sound.sampled.AudioSystem (Class)

The AudioSystem class acts as the entry point to the sampled-audio system resources. This class lets you query and access the mixers that are installed on the system.

3.16.1.1.20 javax.swing.JFrame (Class)

Java class that displays a frame window.

3.16.1.1.21 javax.swing.JOptionPane (Class)

This class is used to display popup messages to an end user.

3.16.1.1.22 javax.swing.JTabbedPane (Class)

This class is a component that has tabbed pages, and the user can click on a tab to flip to a certain page.

3.16.1.1.23 javax.swing.table. AbstractTableModel (Class)

This class provides a base implementation of the TableModel interface. This data structure will be used to supply a JTable with data.

3.16.1.1.24 javax.swing.tree. DefaultTreeModel (Class)

This class is the data structure that is used as a foundation for the JTree class.

3.16.1.1.25 javax.swing.tree. MutableTreeNode (Class)

This interface extends the TreeNode interface and provides the ability to add and remove children from nodes. It may be used in a TreeModel.

3.17 Navigator

3.17.1 Class Diagrams

3.17.1.1 NavigatorClasses (Class Diagram)

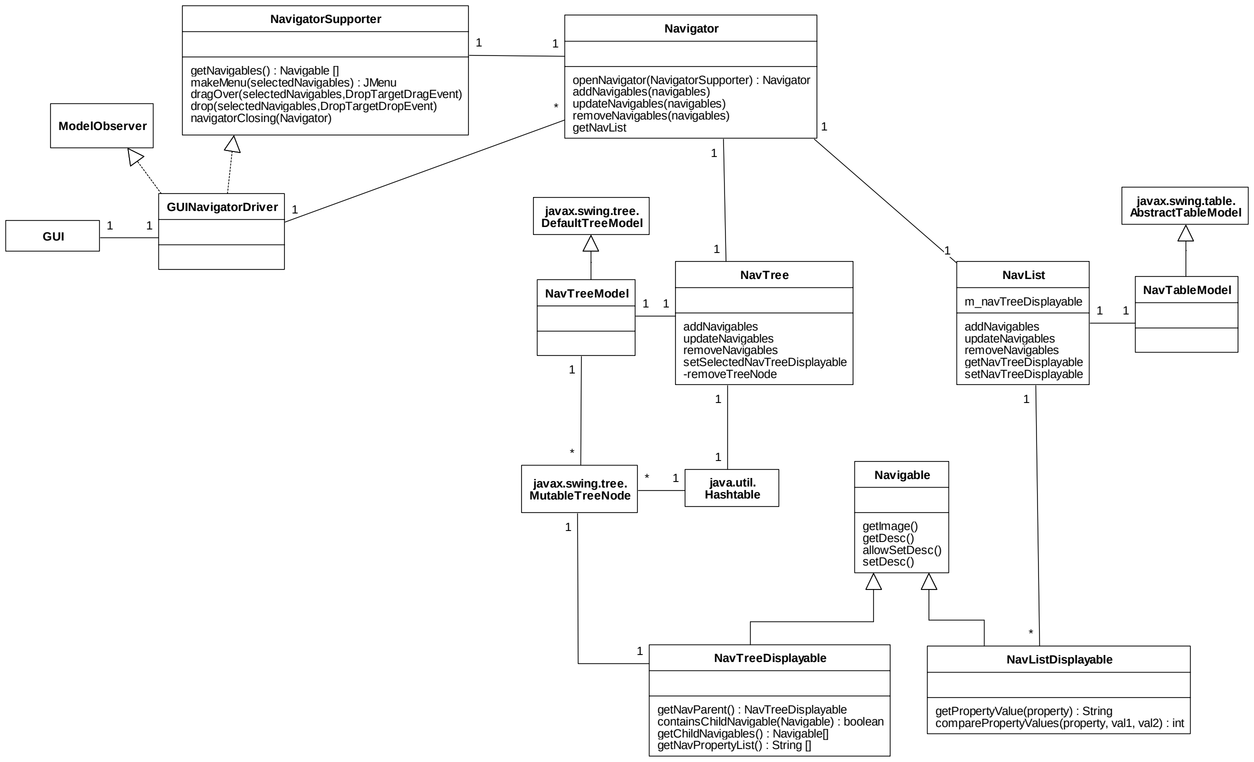

NavigatorSupporter
getNavigables() : Navigable []
makeMenu(selectedNavigables) : JMenu
dragOver(selectedNavigables,DropTargetDragEvent)
drop(selectedNavigables,DropTargetDropEvent)
navigatorClosing(Navigator)
Navigator
openNavigator(NavigatorSupporter) : Navigator
addNavigables(navigables)
updateNavigables(navigables)
removeNavigables(navigables)
getNavList
ModelObserver
GUINavigatorDriver
GUI
javax.swing.tree.
DefaultTreeModel
javax.swing.table.
AbstractTableModel
NavTreeModel
NavTree
addNavigables
updateNavigables
removeNavigables
setSelectedNavTreeDisplayable
-removeTreeNode
NavList
m_navTreeDisplayable
addNavigables
updateNavigables
removeNavigables
getNavTreeDisplayable
setNavTreeDisplayable
NavTableModel
javax.swing.tree.
MutableTreeNode
java.util.
Hashtable
Navigable
getImage()
getDesc()
allowSetDesc()
setDesc()
NavTreeDisplayable
getNavParent() : NavTreeDisplayable
containsChildNavigable(Navigable) : boolean
getChildNavigables() : Navigable[]
getNavPropertyList() : String []
NavListDisplayable
getPropertyValue(property) : String
comparePropertyValues(property, val1, val2) : int
1
1
*
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
*
*
1
1
1
1
*

Figure 205. NavigatorClasses (Class Diagram)

3.17.1.1.1 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

3.17.1.1.2 GUINavigatorDriver (Class)

This class handles all of the Navigator-specific functionality for the GUI.

3.17.1.1.3 java.util. Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

3.17.1.1.4 javax.swing.table. AbstractTableModel (Class)

This class provides a base implementation of the TableModel interface. This data structure will be used to supply a JTable with data.

3.17.1.1.5 javax.swing.tree. DefaultTreeModel (Class)

This class is the data structure that is used as a foundation for the JTree class.

3.17.1.1.6 javax.swing.tree. MutableTreeNode (Class)

This interface extends the TreeNode interface and provides the ability to add and remove children from nodes. It may be used in a TreeModel.

3.17.1.1.7 ModelObserver (Class)

This interface must be implemented by any object that would like to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

3.17.1.1.8 Navigable (Class)

This interface will be implemented by any class that supports the Navigator on either the left or right side (the tree or list view). This includes the functionality common to both the tree and list.

3.17.1.1.9 Navigator (Class)

This class represents one instance of the Navigator window. It supplies methods for opening the Navigator window and for maintaining the collection of Navigables after the Navigator is open.

3.17.1.1.10 NavigatorSupporter (Class)

This interface must be implemented by any subsystem that supports invoking the Navigator. It must be able to supply the Navigable objects, and also can support user interaction with the selected Navigable objects through menus and drag/drop.

3.17.1.1.11 NavList (Class)

This class represents the right hand side of the Navigator window (the list or report). It contains functionality for changing the NavTreeDisplayable to refill the list, and also for maintaining the Navigables in the list after the Navigables belonging to the NavTreeDisplayable are already displayed.

3.17.1.1.12 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

3.17.1.1.13 NavTableModel (Class)

This class will serve as the data structure for the right hand side of the Navigator, and will be the foundation of the JTable that will display the data stored in the model.

3.17.1.1.14 NavTree (Class)

This class represents the left-hand side of the Navigator window - the tree view. It contains functionality for maintaining the NavTreeDisplayable objects that are in the tree.

3.17.1.1.15 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

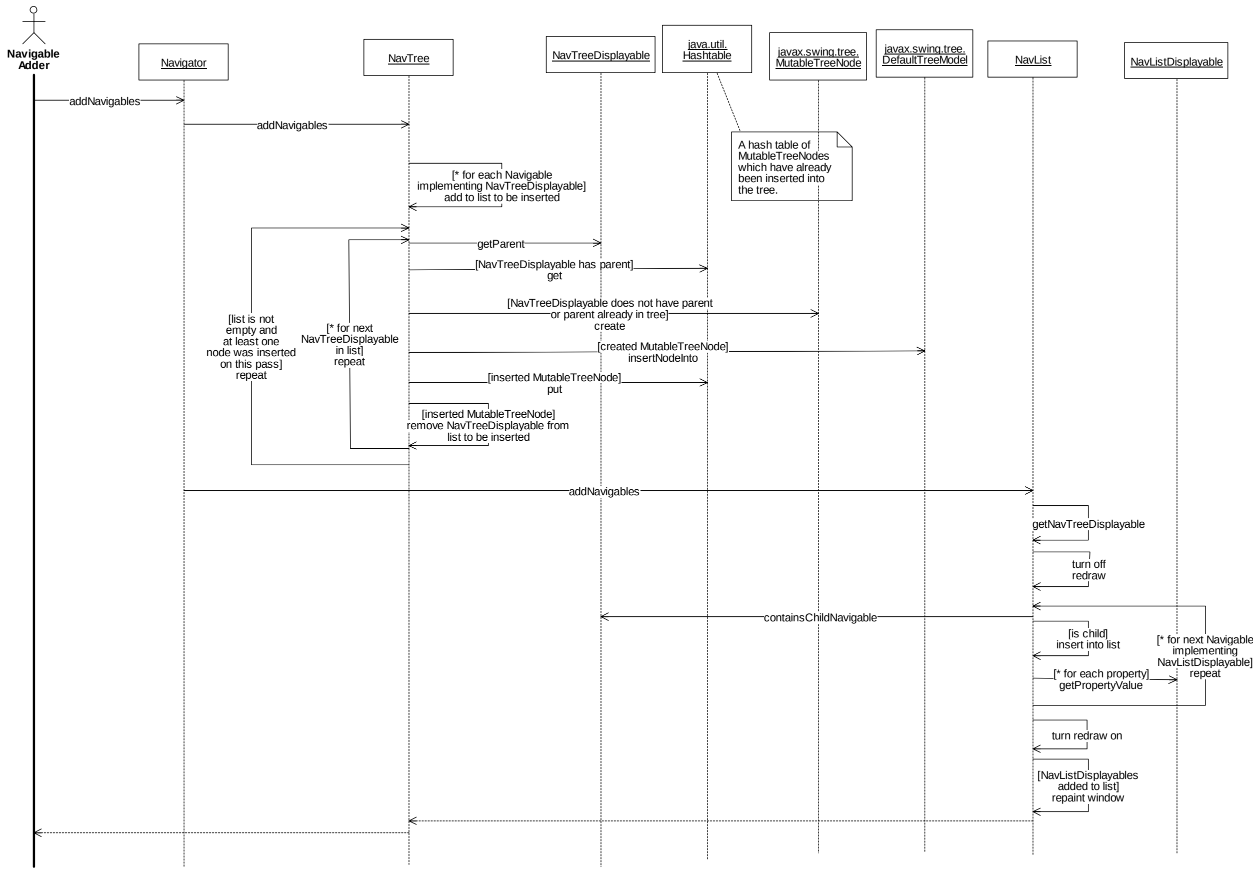
3.17.1.1.16 NavTreeModel (Class)

This class will provide the data structure that will support the tree structure on the left hand side of the Navigator.

3.17.2 Sequence Diagrams

3.17.2.1 Navigator:AddNavigables (Sequence Diagram)

This diagram shows what happens in the Navigator when Navigable objects are added. First, the Navigables are passed to the NavTree. The NavTree will then build a list of any NavTreeDisplayables to add. For each element in the list, it checks the hash table to determine whether the parent (if any) is already in the tree. If the parent is in the tree or there is no parent, a new MutableTreeNode will be created and inserted into the DefaultTreeModel, and the NavTreeDisplayable will be put into the hash table. Each NavTreeDisplayable that is added to the tree is removed from the list to be inserted. As long as one or more nodes were inserted during a given pass through the list, another pass is attempted (for the next level of the tree). Then the Navigables are added to the NavList. This will check each Navigable to see if it is a NavListDisplayable and if its parent is the selected NavTreeDisplayable. If both are true, the NavListDisplayable will be added to the list.



Navigable Adder
Navigator
NavTree
NavTreeDisplayable
java.util.Hashtable
javax.swing.tree.MutableTreeNode
javax.swing.tree.DefaultTreeModel
NavList
NavListDisplayable
addNavigables
addNavigables
A hash table of MutableTreeNodes which have already been inserted into the tree.
[* for each Navigable implementing NavTreeDisplayable] add to list to be inserted
getParent
[NavTreeDisplayable has parent] get
[NavTreeDisplayable does not have parent or parent already in tree] create
[list is not empty and at least one node was inserted on this pass] repeat
[* for next NavTreeDisplayable in list] repeat
[created MutableTreeNode] insertNodeInto
[inserted MutableTreeNode] put
[inserted MutableTreeNode] remove NavTreeDisplayable from list to be inserted
addNavigables
getNavTreeDisplayable
turn off redraw
containsChildNavigable
[is child] insert into list
[* for next Navigable implementing NavListDisplayable] repeat
[* for each property] getPropertyValue
turn redraw on
[NavListDisplayables added to list] repaint window

Figure 206. Navigator:AddNavigables (Sequence Diagram)

3.17.2.2 Navigator:Initialize (Sequence Diagram)

This diagram shows how the Navigator is initialized. The openNavigator method will create a new Navigator window and the tree and list views. The Navigator will then query the NavigatorSupporter to provide it with all Navigable objects. The Navigables are added to the NavTree (see the Navigator:AddNavigables diagram for details). Then the root node is set as the selected node in the NavTree. See the Navigator:TreeSelectionChange sequence diagram for details on the effects of this.

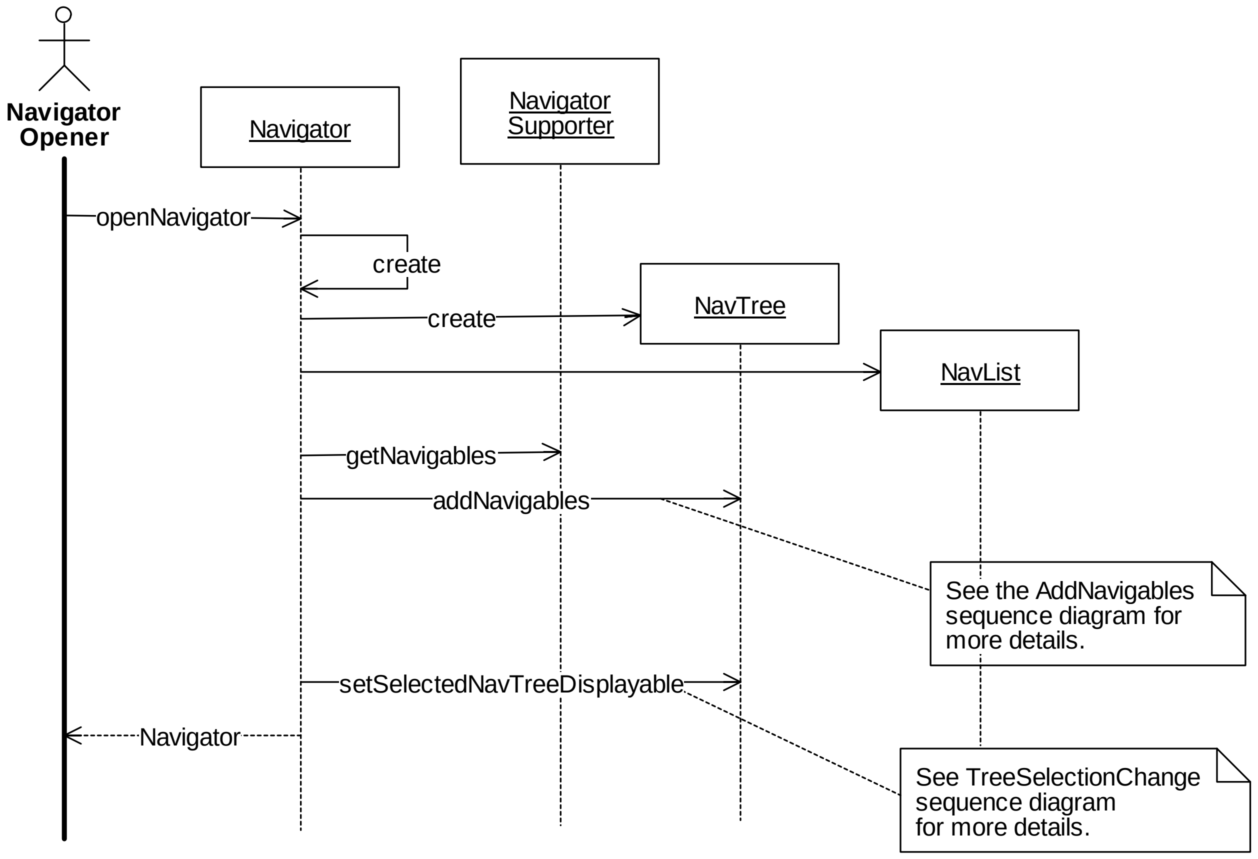

Navigator Opener
Navigator
Navigator Supporter
openNavigator
create
create
NavTree
NavList
getNavigables
addNavigables
See the AddNavigables sequence diagram for more details.
setSelectedNavTreeDisplayable
Navigator
See TreeSelectionChange sequence diagram for more details.

Figure 207. Navigator:Initialize (Sequence Diagram)

3.17.2.3 Navigator:RemoveNavigables (Sequence Diagram)

This diagram shows how Navigables are removed from the Navigator. Each NavTreeDisplayable to removed causes removeTreeNode to be called. This is a recursive call, which calls removeTreeNode first on each of its children. The children are removed first so that every tree node below the current node is cleaned out of the hash table. If the NavList is displaying the children of the node that is being destroyed, then we set the NavTreeDisplayable in the list to the parent. Then the NavTreeDisplayable is removed from the hash table and also from its parent. The Navigables to be removed are then passed to the NavList, which removes and NavListDisplayables in the list matching any of the Navigables to be removed.

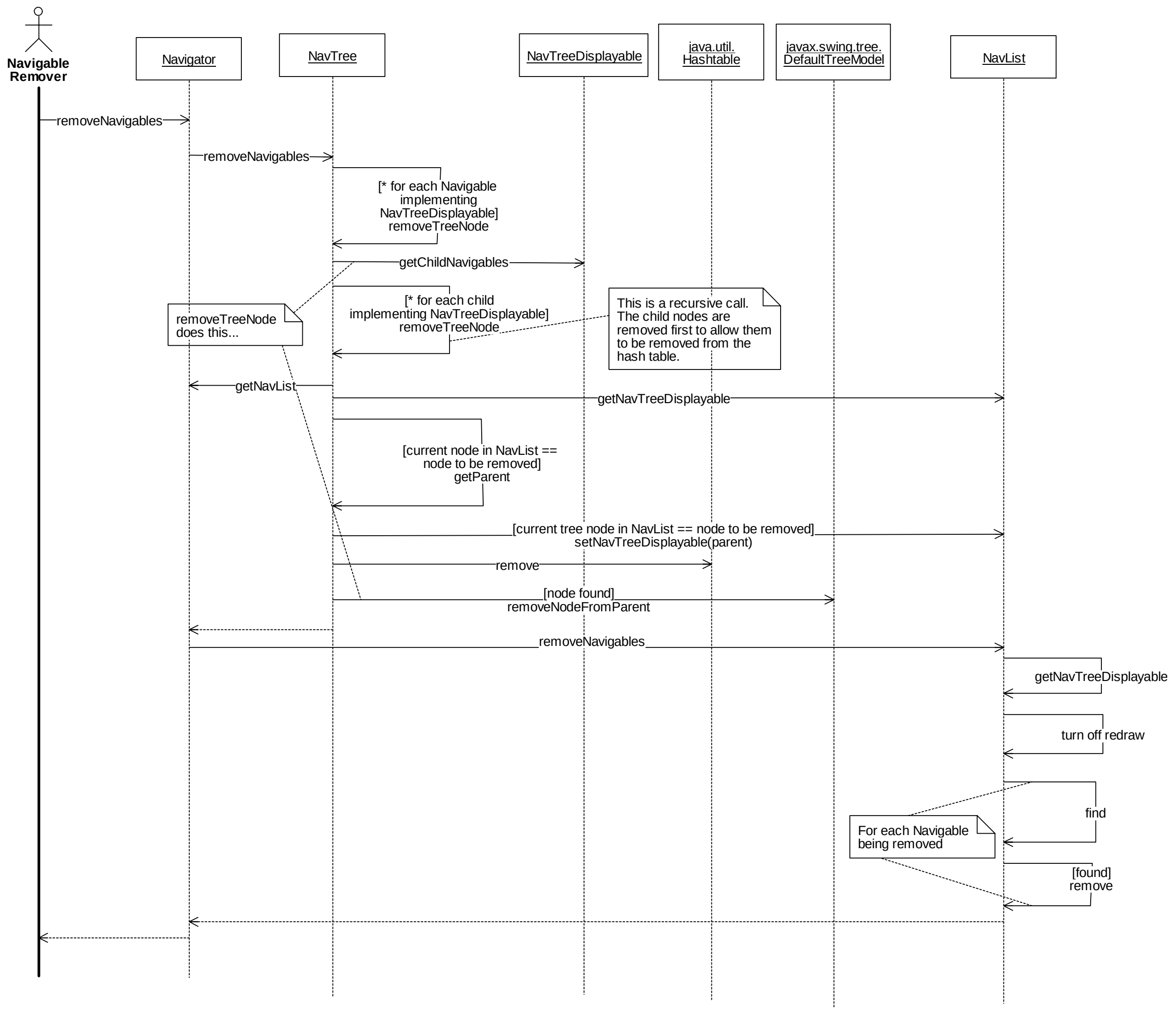

Navigable Remover
Navigator
NavTree
NavTreeDisplayable
java.util. Hashtable
javax.swing.tree. DefaultTreeModel
NavList
removeNavigables
removeNavigables
[* for each Navigable implementing NavTreeDisplayable] removeTreeNode
getChildNavigables
[* for each child implementing NavTreeDisplayable] removeTreeNode
removeTreeNode does this...
This is a recursive call. The child nodes are removed first to allow them to be removed from the hash table.
getNavList
getNavTreeDisplayable
[current node in NavList == node to be removed] getParent
[current tree node in NavList == node to be removed] setNavTreeDisplayable(parent)
remove
[node found] removeNodeFromParent
removeNavigables
getNavTreeDisplayable
turn off redraw
find
For each Navigable being removed
[found] remove

Figure 208. Navigator:RemoveNavigables (Sequence Diagram)

3.17.2.4 Navigator:TreeSelectionChange (Sequence Diagram)

This diagram shows what happens when a tree selection change takes place. The NavTree calls the NavList and sets the NavTreeDisplayable. This will cause all objects to be removed from the NavList. The NavList will ask the new NavTreeDisplayable for its properties (columns). Then the NavList will ask the NavTreeDisplayable for its children, which will all be inserted into the list. Each item inserted will be called for each column/property to supply the property value.

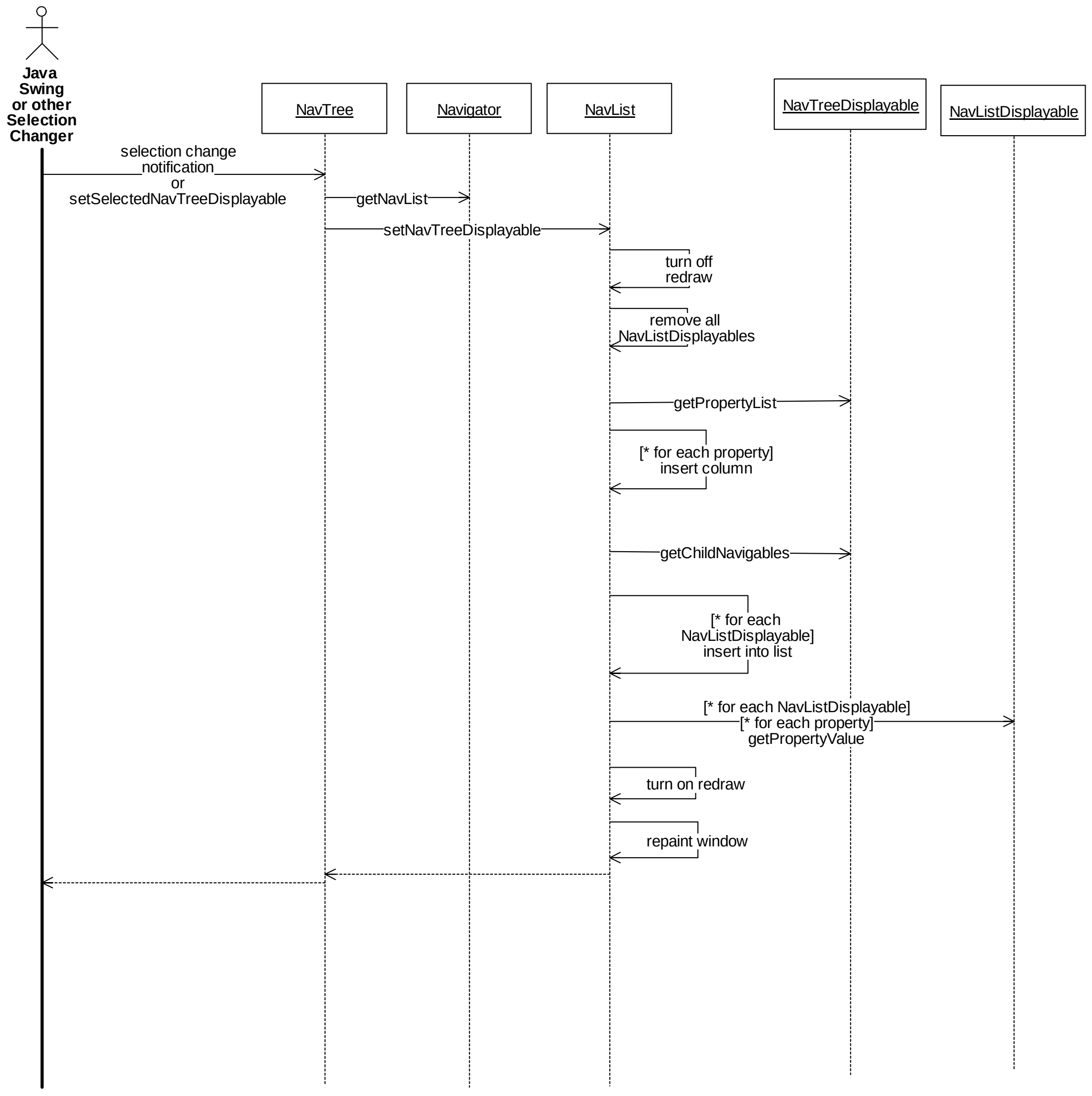

Java Swing or other Selection Changer
NavTree
Navigator
NavList
NavTreeDisplayable
NavListDisplayable
selection change notification or setSelectedNavTreeDisplayable
getNavList
setNavTreeDisplayable
turn off redraw
remove all NavListDisplayables
getPropertyList
[* for each property] insert column
getChildNavigables
[* for each NavListDisplayable] insert into list
[* for each NavListDisplayable] [* for each property] getPropertyValue
turn on redraw
repaint window

Figure 209. Navigator:TreeSelectionChange (Sequence Diagram)

[DCE:417]

3.18 Shazam Utility

3.18.1 Class Diagrams

3.18.1.1 SHAZAMUtility (Class Diagram)

This diagram shows SHAZAM related classes that are shared between the server and the GUI.

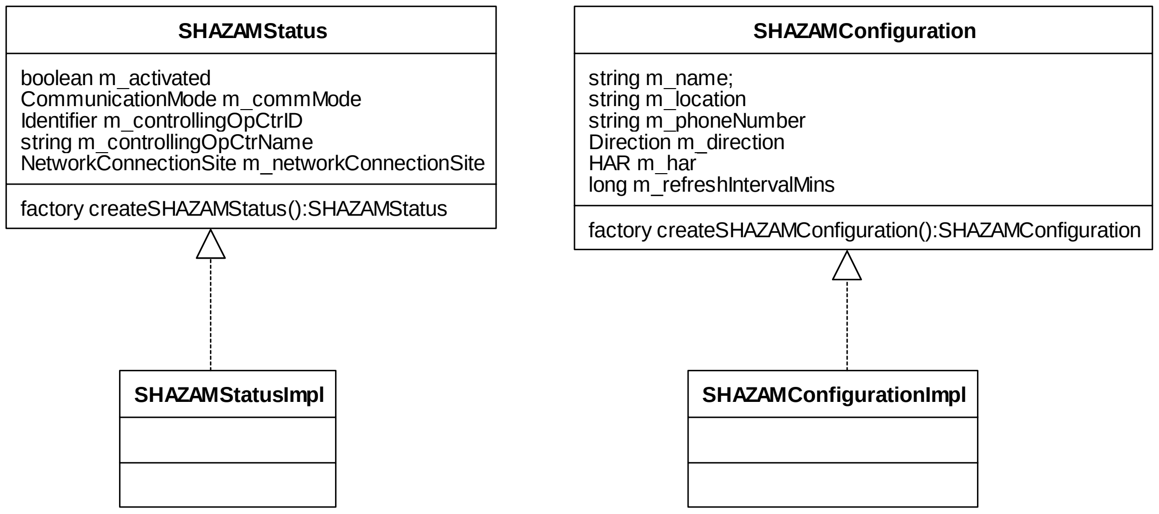

SHAZAMStatus
boolean m_activated
CommunicationMode m_commMode
Identifier m_controllingOpCtrID
string m_controllingOpCtrName
NetworkConnectionSite m_networkConnectionSite
factory createSHAZAMStatus():SHAZAMStatus
SHAZAMConfiguration
string m_name;
string m_location
string m_phoneNumber
Direction m_direction
HAR m_har
long m_refreshIntervalMins
factory createSHAZAMConfiguration():SHAZAMConfiguration
SHAZAMStatusImpl
SHAZAMConfigurationImpl

Figure 210. SHAZAMUtility (Class Diagram)

3.18.1.1.1 SHAZAMConfiguration (Class)

This class contains data that specifies the configuration of a SHAZAM device.

3.18.1.1.2 SHAZAMConfigurationImpl (Class)

This class provides an implementation of the SHAZAMConfiguration valuetype as defined in the IDL. This class provides access to values relating to the configuration of a SHAZAM.

3.18.1.1.3 SHAZAMStatus (Class)

This class contains the current status of a SHAZAM device.

3.18.1.1.4 SHAZAMStatusImpl (Class)

This class implements the SHAZAMStatus valuetype as defined in the IDL. It provides access to values relating to the current status of a SHAZAM.

3.19 System Interfaces

3.19.1 Class Diagrams

3.19.1.1 AudioCommon (Class Diagram)

This class diagram shows the classes relating to Audio.

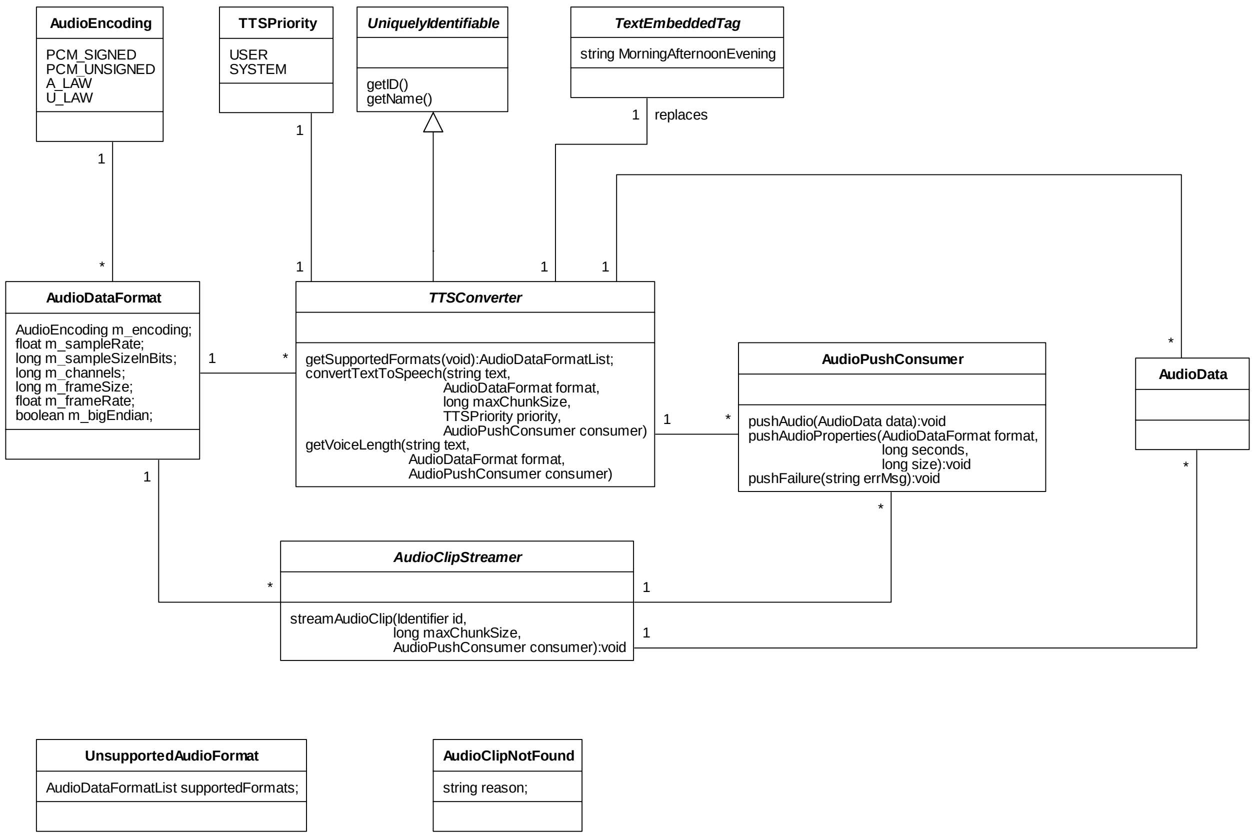

AudioEncoding
PCM_SIGNED
PCM_UNSIGNED
A_LAW
U_LAW
TTSPriority
USER
SYSTEM
UniquelyIdentifiable
getID()
getName()
TextEmbeddedTag
string MorningAfternoonEvening
replaces
AudioDataFormat
AudioEncoding m_encoding;
float m_sampleRate;
long m_sampleSizeInBits;
long m_channels;
long m_frameSize;
float m_frameRate;
boolean m_bigEndian;
TTSConverter
getSupportedFormats(void):AudioDataFormatList;
convertTextToSpeech(string text, AudioDataFormat format, long maxChunkSize, TTSPriority priority, AudioPushConsumer consumer)
getVoiceLength(string text, AudioDataFormat format, AudioPushConsumer consumer)
AudioPushConsumer
pushAudio(AudioData data):void
pushAudioProperties(AudioDataFormat format, long seconds, long size):void
pushFailure(string errMsg):void
AudioData
AudioClipStreamer
streamAudioClip(Identifier id, long maxChunkSize, AudioPushConsumer consumer):void
UnsupportedAudioFormat
AudioDataFormatList supportedFormats;
AudioClipNotFound
string reason;

Figure 211. AudioCommon (Class Diagram)

3.19.1.1.1 AudioClipNotFound (Class)

This exception is thrown by an AudioClipStreamer if asked to push an audio clip which it cannot find.

3.19.1.1.2 AudioClipStreamer (Class)

This interface is implemented by objects that can push a previously stored audio clip given its ID. The audio data is pushed via the AudioPushConsumer supplied by the user of this interface.

3.19.1.1.3 AudioData (Class)

This typedef is a sequence of bytes that contain audio data. This data is used in conjunction with AudioDataFormat to decode the data into voice.

3.19.1.1.4 AudioDataFormat (Class)

This struct specifies the format of audio data.

3.19.1.1.5 AudioEncoding (Class)

This enum defines the supported types of encoding for audio data.

3.19.1.1.6 AudioPushConsumer (Class)

This interface is implemented by objects that wish to receive audio data using the push model, where the server pushes the data to the consumer. One call to pushAudioProperties() will always precede any calls to pushAudio().

3.19.1.1.7 TextEmbeddedTag (Class)

This interface defines constants for tags that may be embedded in text that is passed to the TTSConverter. The TTSConverter replaces the tags it finds in text prior to converting the text to speech. The MorningAfternoonEvening tag is replaced with the text ‘morning’ when the conversion takes place between 00:00 and 11:59, ‘afternoon’ from 12:00 through 16:59, and ‘evening’ from 17:00 to 23:59.

3.19.1.1.8 TTSConverter (Class)

This interface represents the Text to Speech converter object that allows text to be passed in and speech to be returned.

3.19.1.1.9 TTSPriority (Class)

This enum defines the types of priorities that can be used when asking the TTSConverter to convert text to speech.

3.19.1.1.10 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.1.11 UnsupportedAudioFormat (Class)

This exception is thrown when a specific AudioDataFormat is requested from an object that does not support the given format.

3.19.1.2 CommLogManagement (Class Diagram)

This Class Diagram shows the classes used for passing information between processes to enable creating, pushing, viewing, and searching Communications Log entries.

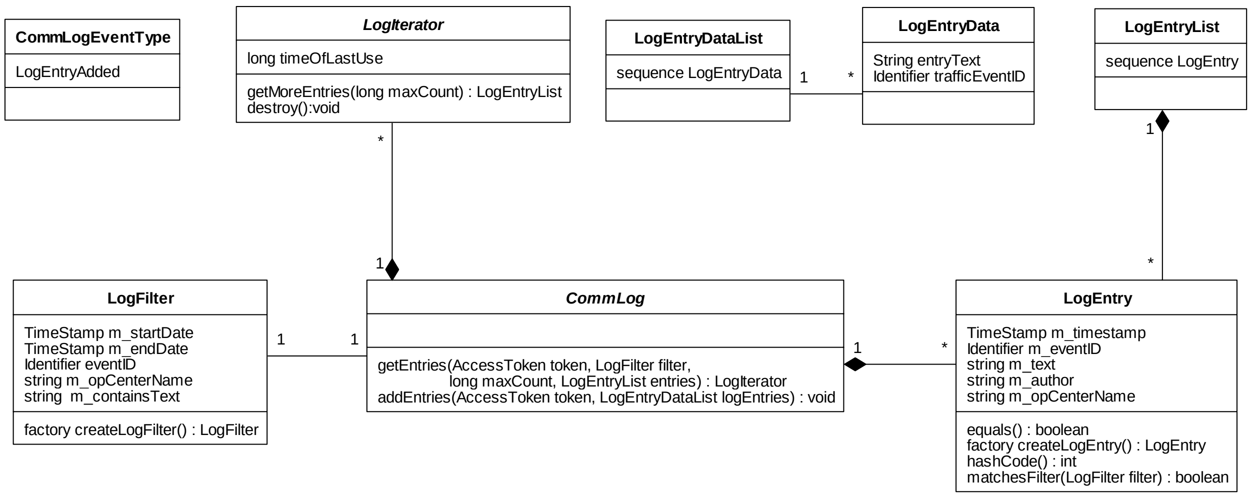


Figure 212. CommLogManagement (Class Diagram)

3.19.1.2.1 CommLog (Class)

This class manages log entries. These can be general Communications Log entries or specific log entries for a specific Traffic Event. This class is the primary interface for the CommLog service. It is used to persist log entries in the CHART II system and retrieve them for review. Log entries can be created directly by users or indirectly as a result of manipulating Traffic Events.

3.19.1.2.2 CommLogEventType (Class)

This enumeration lists the possible events that the CommsLog service may push via the CORBA event service. At present, only one event is defined, the addition of a new LogEntry to the database.

3.19.1.2.3 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.19.1.2.4 LogEntryData (Class)

LogEntryData is a collection of data required to create one Log Entry, consisting of text (the body of the event) and an ID that refers to a Traffic Event, if appropriate.

3.19.1.2.5 LogEntryDataList (Class)

The LogEntryDataList is simply a sequence of LogEntryData objects, each of which contain the data needed to create one Log Entry. Normally each LogEntryDataList will contain only one LogEntryData object, but if the CommLog service is unavailable for a time, it is possible that multiple LogEntryData objects may be queued up for insertion into the database.

3.19.1.2.6 LogEntryList (Class)

The LogEntryList is simply a sequence of LogEntry instances returned to a requesting process in one clump. (Some requests return so much data that data is returned in clumps. The initial request returns a LogIterator from which additional LogEntryList sequences can be requested, in order to complete the entire query.

3.19.1.2.7 LogFilter (Class)

This class is used to specify the criteria to be used when getting entries from the Communications Log. The caller would create an object of this type specifying the criteria that each log entry must match in order to be returned.

3.19.1.2.8 LogIterator (Class)

This class represents an iterator to iterate through a collection of log entries. If a retrieval request results in more data than is reasonable to transmit all at once, one clump of entries is returned at first, together with a LogIterator from which additional data can be requested, repeatedly, until all entries are returned or the user cancels the operation.

3.19.1.3 Common (Class Diagram)

This class diagram shows classes used by multiple modules.

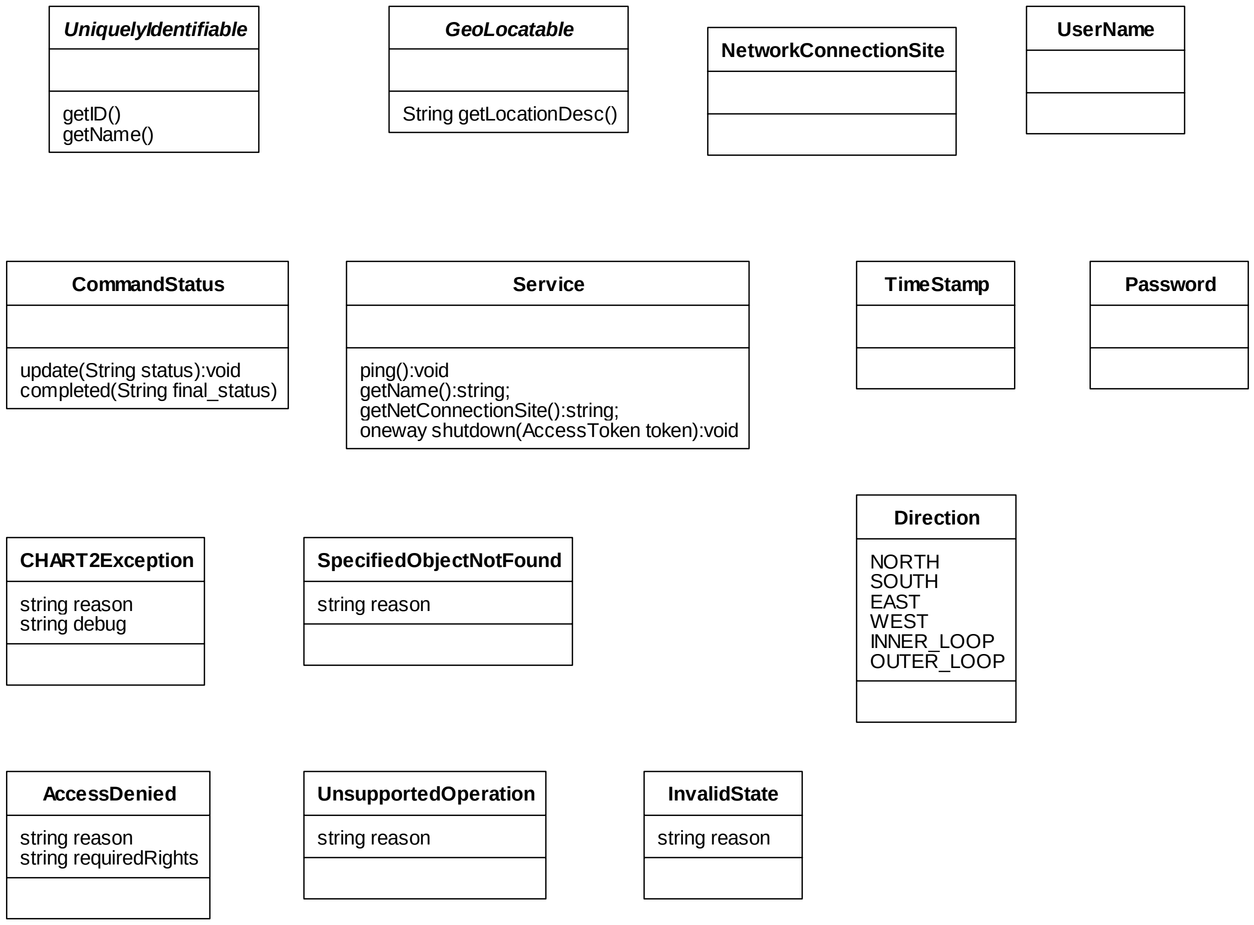


Figure 213. Common (Class Diagram)

3.19.1.3.1 AccessDenied (Class)

This class represents an access denied, or “no rights” failure.

3.19.1.3.2 CHART2Exception (Class)

Generic exception class for the CHART2 system. This class can be used for throwing very generic exceptions which require no special processing by the client. It supports a reason string that may be shown to any user and a debug string that will contain detailed information useful in determining the cause of the problem.

3.19.1.3.3 CommandStatus (Class)

The CommandStatus CORBA interface is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

3.19.1.3.4 Direction (Class)

This enumeration defines direction of travel.

3.19.1.3.5 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.19.1.3.6 InvalidState (Class)

This exception is thrown when an operation is attempted on an object that is not in a valid state to perform the operation.

3.19.1.3.7 NetworkConnectionSite (Class)

The NetworkConnectionSite class contains a string that is used to specify where a service is running. This field is useful for administrators in debugging problems should an object become “software comm failed”. It is included in the Chart2DMSStatus.

3.19.1.3.8 Password (Class)

Typedef used to define the type of a Password.

3.19.1.3.9 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

3.19.1.3.10 SpecifiedObjectNotFound (Class)

Exception used to indicate that an operation was attempted that involves a secondary object that cannot be found by the invoked object.

3.19.1.3.11 TimeStamp (Class)

This typedef defines the type of TimeStamp fields.

3.19.1.3.12 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.3.13 UnsupportedOperation (Class)

This exception is used to indicate that an operation is not supported by the object on which it is called.

3.19.1.3.14 UserName (Class)

This typedef defines the type of UserName fields used in system interfaces.

3.19.1.4 DeviceManagement (Class Diagram)

This class diagram shows device interfaces that are common among devices.

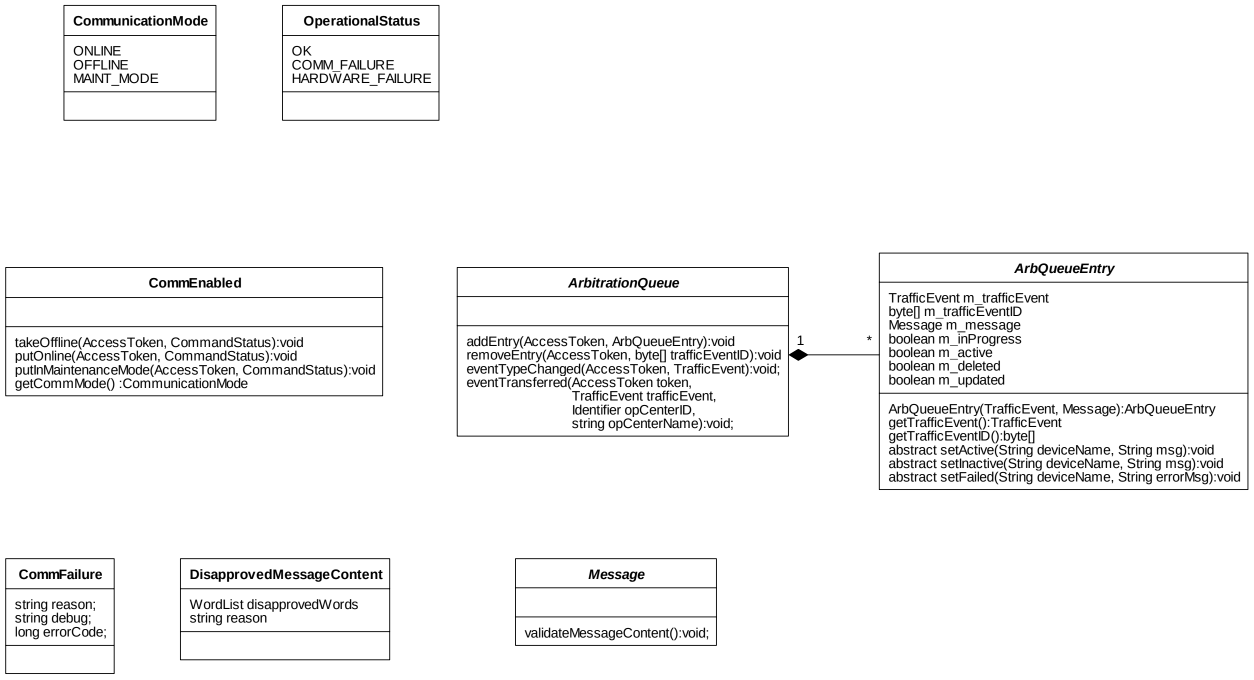

CommunicationMode
ONLINE
OFFLINE
MAINT_MODE
OperationalStatus
OK
COMM_FAILURE
HARDWARE_FAILURE
CommEnabled
takeOffline(AccessToken, CommandStatus):void
putOnline(AccessToken, CommandStatus):void
putInMaintenanceMode(AccessToken, CommandStatus):void
getCommMode() :CommunicationMode
ArbitrationQueue
addEntry(AccessToken, ArbQueueEntry):void
removeEntry(AccessToken, byte[] trafficEventID):void
eventTypeChanged(AccessToken, TrafficEvent):void;
eventTransferred(AccessToken token,
TrafficEvent trafficEvent,
Identifier opCenterID,
string opCenterName):void;
1
*
ArbQueueEntry
TrafficEvent m_trafficEvent
byte[] m_trafficEventID
Message m_message
boolean m_inProgress
boolean m_active
boolean m_deleted
boolean m_updated
ArbQueueEntry(TrafficEvent, Message):ArbQueueEntry
getTrafficEvent():TrafficEvent
getTrafficEventID():byte[]
abstract setActive(String deviceName, String msg):void
abstract setInactive(String deviceName, String msg):void
abstract setFailed(String deviceName, String errorMsg):void
CommFailure
string reason;
string debug;
long errorCode;
DisapprovedMessageContent
WordList disapprovedWords
string reason
Message
validateMessageContent():void;

Figure 214. DeviceManagement (Class Diagram)

3.19.1.4.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue’s slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue’s automatic processing interfere with the maintenance activities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue performs no special processing when resumed because the queue cleans itself when interrupted and does not allow new entries while interrupted.

3.19.1.4.2 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.19.1.4.3 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.19.1.4.4 CommFailure (Class)

This exception is to be thrown when an error is detected connecting to or communicating with a device.

3.19.1.4.5 CommunicationMode (Class)

The CommunicationMode class enumerations the modes of operation for a DMS: ONLINE, OFFLINE, and MAINT_MODE. The DMSStatus class contains a value of this type.

3.19.1.4.6 DisapprovedMessageContent (Class)

This exception is thrown when a text message to be put on a device contains words that are not approved. This exception is also thrown if an attempt is made to put the device in an invalid display state, such as putting the Beacons ON for a blank DMS.

3.19.1.4.7 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.19.1.4.8 OperationalStatus (Class)

The OperationalStatus class enumerates the types of operational status a DMS can have: OK (normal mode), COMM_FAILURE (no communications to the device), or HARDWARE_FAILURE (device is reachable but is reporting a hardware failure). The DMSStatus class contains a value of this type.

3.19.1.5 DictionaryManagement (Class Diagram)

This class diagram shows the interfaces used for the dictionaries.

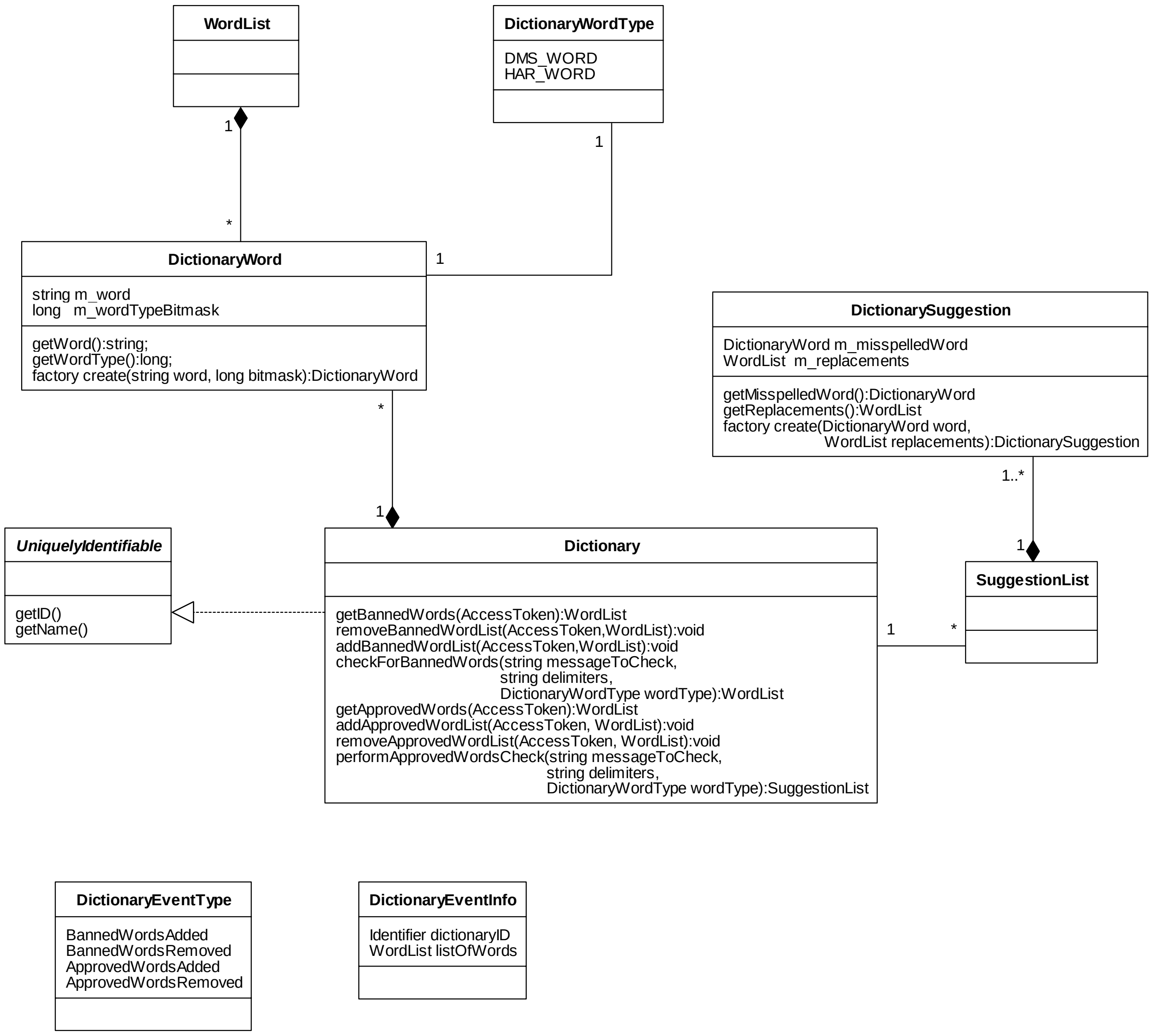

WordList
DictionaryWordType
DMS_WORD
HAR_WORD
1
1
*
DictionaryWord
string m_word
long m_wordTypeBitmask
getWord():string;
getWordType():long;
factory create(string word, long bitmask):DictionaryWord
1
DictionarySuggestion
DictionaryWord m_misspelledWord
WordList m_replacements
getMisspelledWord():DictionaryWord
getReplacements():WordList
factory create(DictionaryWord word, WordList replacements):DictionarySuggestion
*
1..*
1
1
UniquelyIdentifiable
getID()
getName()
Dictionary
getBannedWords(AccessToken):WordList
removeBannedWordList(AccessToken,WordList):void
addBannedWordList(AccessToken,WordList):void
checkForBannedWords(string messageToCheck, string delimiters, DictionaryWordType wordType):WordList
getApprovedWords(AccessToken):WordList
addApprovedWordList(AccessToken, WordList):void
removeApprovedWordList(AccessToken, WordList):void
performApprovedWordsCheck(string messageToCheck, string delimiters, DictionaryWordType wordType):SuggestionList
1
*
SuggestionList
DictionaryEventType
BannedWordsAdded
BannedWordsRemoved
ApprovedWordsAdded
ApprovedWordsRemoved
DictionaryEventInfo
Identifier dictionaryID
WordList listOfWords

Figure 215. DictionaryManagement (Class Diagram)

3.19.1.5.1 Dictionary (Class)

The Dictionary IDL interface provides functionality to add, delete and check for words that are approved or banned from being used in a CHART2 messaging device. Examples of messaging devices are DMS, HAR, etc.

3.19.1.5.2 DictionaryEventInfo (Class)

This interface encapsulates the data that is passed with a dictionary CORBA event. It contains information identifying the dictionary, and the list of words affected by the event.

3.19.1.5.3 DictionaryEventType (Class)

This represents the enumerations used for the different CORBA event types applicable to the dictionary module.

3.19.1.5.4 DictionarySuggestion (Class)

A DictionarySuggestion represents a list of suggested words that may be used as a substitute for the word that could not be found in the approved words dictionary database.

3.19.1.5.5 DictionaryWord (Class)

A DictionaryWord represents a word in the chart2 dictionary. It contains information that qualifies the type of devices that the word applies to.

3.19.1.5.6 DictionaryWordType (Class)

3.19.1.5.7 SuggestionList (Class)

This interface represents the IDL sequence typedef for the DictionarySuggestion.

3.19.1.5.8 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.5.9 WordList (Class)

This interface represents the IDL sequence typedef for the DictionaryWord.

3.19.1.6 DMSControl (Class Diagram)

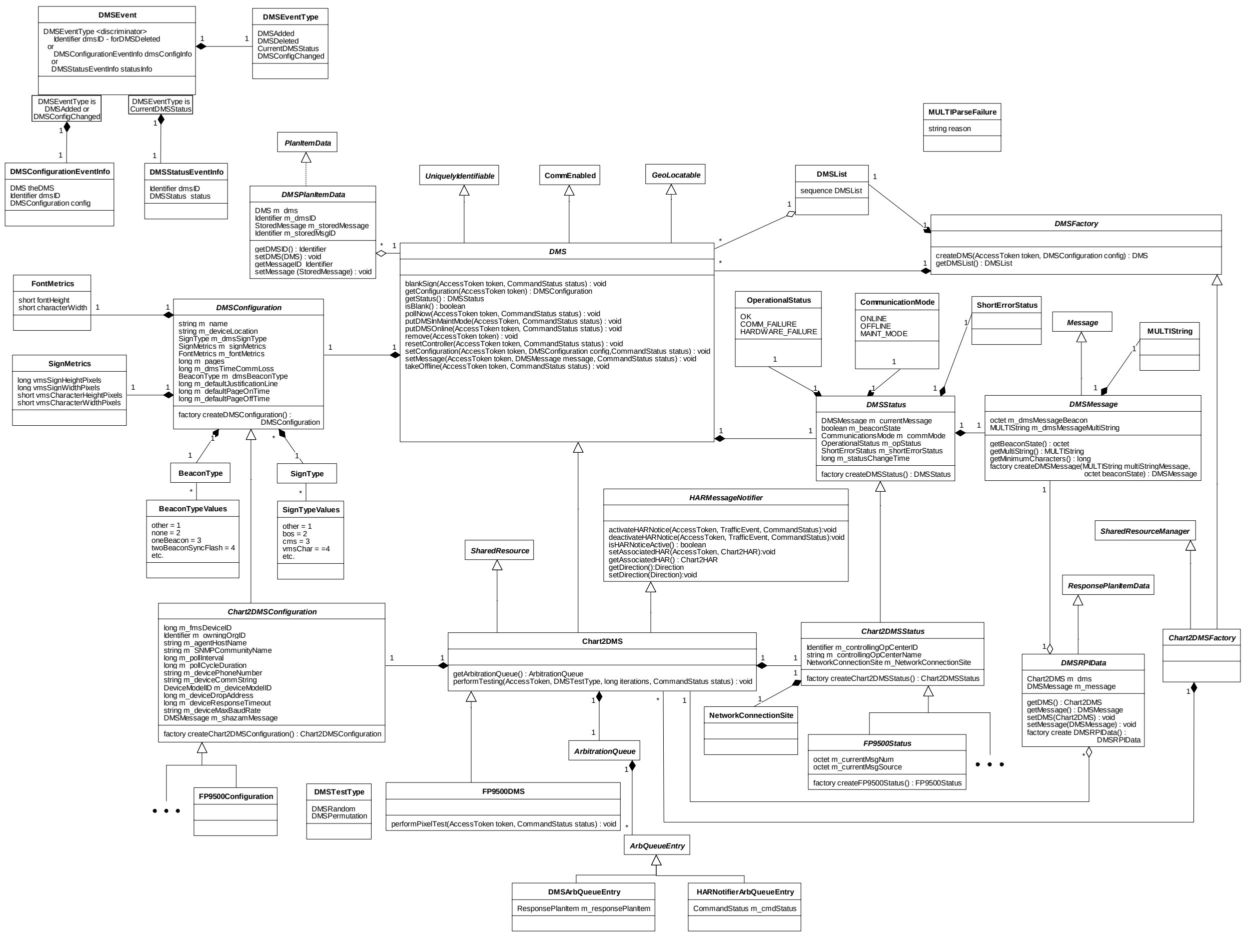

DMSEvent
DMSEventType <discriminator>
Identifier dmsID - forDMSDeleted
or
DMSConfigurationEventInfo dmsConfigInfo
or
DMSStatusEventInfo statusInfo
DMSEventType
DMSAdded
DMSDeleted
CurrentDMSStatus
DMSConfigChanged
DMSEventType is DMSAdded or DMSConfigChanged
DMSEventType is CurrentDMSStatus
DMSConfigurationEventInfo
DMS theDMS
Identifier dmsID
DMSConfiguration config
DMSStatusEventInfo
Identifier dmsID
DMSStatus status
PlanItemData
DMSPlanItemData
DMS m_dms
Identifier m_dmsID
StoredMessage m_storedMessage
Identifier m_storedMsgID
getDMSID() : Identifier
setDMS(DMS) : void
getMessageID Identifier
setMessage (StoredMessage) : void
UniquelyIdentifiable
CommEnabled
GeoLocatable
DMSList
sequence DMSList
MULTIParseFailure
string reason
DMSFactory
createDMS(AccessToken token, DMSConfiguration config) : DMS
getDMSList() : DMSList
DMS
blankSign(AccessToken token, CommandStatus status) : void
getConfiguration(AccessToken token) : DMSConfiguration
getStatus() : DMSStatus
isBlank() : boolean
pollNow(AccessToken token, CommandStatus status) : void
putDMSInMaintMode(AccessToken, CommandStatus status) : void
putDMSOnline(AccessToken token, CommandStatus status) : void
remove(AccessToken token) : void
resetController(AccessToken token, CommandStatus status) : void
setConfiguration(AccessToken token, DMSConfiguration config,CommandStatus status) : void
setMessage(AccessToken token, DMSMessage message, CommandStatus status) : void
takeOffline(AccessToken token, CommandStatus status) : void
FontMetrics
short fontHeight
short characterWidth
SignMetrics
long vmsSignHeightPixels
long vmsSignWidthPixels
short vmsCharacterHeightPixels
short vmsCharacterWidthPixels
DMSConfiguration
string m_name
string m_deviceLocation
SignType m_dmsSignType
SignMetrics m_signMetrics
FontMetrics m_fontMetrics
long m_pages
long m_dmsTimeCommLoss
BeaconType m_dmsBeaconType
long m_defaultJustificationLine
long m_defaultPageOnTime
long m_defaultPageOffTime
factory createDMSConfiguration() : DMSConfiguration
BeaconType
SignType
BeaconTypeValues
other = 1
none = 2
oneBeacon = 3
twoBeaconSyncFlash = 4
etc.
SignTypeValues
other = 1
bos = 2
cms = 3
vmsChar = =4
etc.
OperationalStatus
OK
COMM_FAILURE
HARDWARE_FAILURE
CommunicationMode
ONLINE
OFFLINE
MAINT_MODE
ShortErrorStatus
Message
MULTIString
DMSStatus
DMSMessage m_currentMessage
boolean m_beaconState
CommunicationsMode m_commMode
OperationalStatus m_opStatus
ShortErrorStatus m_shortErrorStatus
long m_statusChangeTime
factory createDMSStatus() : DMSStatus
DMSMessage
octet m_dmsMessageBeacon
MULTIString m_dmsMessageMultiString
getBeaconState() : octet
getMultiString() : MULTIString
getMinimumCharacters() : long
factory createDMSMessage(MULTIString multiStringMessage, octet beaconState) : DMSMessage
HARMessageNotifier
activateHARNotice(AccessToken, TrafficEvent, CommandStatus):void
deactivateHARNotice(AccessToken, TrafficEvent, CommandStatus):void
isHARNoticeActive() : boolean
setAssociatedHAR(AccessToken, Chart2HAR):void
getAssociatedHAR() : Chart2HAR
getDirection():Direction
setDirection(Direction):void
SharedResource
SharedResourceManager
ResponsePlanItemData
Chart2DMSConfiguration
long m_fmsDeviceID
Identifier m_owningOrgID
string m_agentHostName
string m_SNMPCommunityName
long m_pollInterval
long m_pollCycleDuration
string m_devicePhoneNumber
string m_deviceCommString
DeviceModelID m_deviceModelID
long m_deviceDropAddress
long m_deviceResponseTimeout
string m_deviceMaxBaudRate
DMSMessage m_shazamMessage
factory createChart2DMSConfiguration() : Chart2DMSConfiguration
Chart2DMS
getArbitrationQueue() : ArbitrationQueue
performTesting(AccessToken, DMSTestType, long iterations, CommandStatus status) : void
Chart2DMSStatus
Identifier m_controllingOpCenterID
string m_controllingOpCenterName
NetworkConnectionSite m_NetworkConnectionSite
factory createChart2DMSStatus() : Chart2DMSStatus
DMSRPIData
Chart2DMS m_dms
DMSMessage m_message
getDMS() : Chart2DMS
getMessage() : DMSMessage
setDMS(Chart2DMS) : void
setMessage(DMSMessage) : void
factory create DMSRPIData() : DMSRPIData
Chart2DMSFactory
NetworkConnectionSite
ArbitrationQueue
FP9500Status
octet m_currentMsgNum
octet m_currentMsgSource
factory createFP9500Status() : FP9500Status
FP9500Configuration
DMSTestType
DMSRandom
DMSPermutation
FP9500DMS
performPixelTest(AccessToken token, CommandStatus status) : void
ArbQueueEntry
DMSArbQueueEntry
ResponsePlanItem m_responsePlanItem
HARNotifierArbQueueEntry
CommandStatus m_cmdStatus
• • •

Figure 216. DMSControl (Class Diagram)

3.19.1.6.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue performs no special processing when resumed because the queue cleans itself when interrupted and does not allow new entries while interrupted.

3.19.1.6.2 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.19.1.6.3 BeaconType (Class)

The BeaconType class defines the beacon type for a DMS. Its values are defined by the BeaconTypeValues class. It is a part of a DMSConfiguration object.

3.19.1.6.4 BeaconTypeValues (Class)

The BeaconTypeValues class enumerates the various beacon types used on DMS devices (number of beacons and whether and in what manner they flash).

3.19.1.6.5 Chart2DMS (Class)

The Chart2DMS class extends the DMS interface and defines a more detailed interface to be used in manipulating the Chart II-specific DMS objects within Chart II. It provides a method for getting the DMSArbitrationQueue for a Chart II DMS, which can then be used by traffic events to provide input as to what each traffic event desires to be on the sign. It also provides a method to perform testing on a sign. This method can be extended by derived classes for specific models of signs, which know how to perform certain types of testing on their specific model of sign. Chart II business rules include concepts such as shared resources, arbitration queues, and linking devices usage to traffic events, concepts which go beyond what would be industry-standard DMS control.

3.19.1.6.6 Chart2DMSConfiguration (Class)

The Chart2DMSConfiguration class is an abstract class which extends the DMSConfiguration class to provide configuration information specific to Chart II processing. Such information includes how to contact the sign under Chart II software control, the default SHAZAM message for using the sign as a HAR Notifier, and the owning organization. Such data extends beyond what would be industry-standard configuration information for a DMS.

3.19.1.6.7 Chart2DMSFactory (Class)

The Chart2DMSFactory class extends the DMSFactory interface to provide additional Chart II specific capability. This factory creates Chart2DMS objects (extensions of DMS objects). It implements SharedResourceManager capbility control DMS objects as shared resources.

3.19.1.6.8 Chart2DMSStatus (Class)

The Chart2DMSStatus class is an abstract class that extends the DMSStatus class to provide status information specific to Chart II processing, such as information on the controlling operations center for the sign. This data extends beyond what would be industry-standard status information for a DMS.

3.19.1.6.9 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.19.1.6.10 CommunicationMode (Class)

The CommunicationMode class enumerations the modes of operation for a DMS: ONLINE, OFFLINE, and MAINT_MODE. The DMSStatus class contains a value of this type.

3.19.1.6.11 DMS (Class)

The DMS class defines an interface to be used in manipulating Dynamic Message Sign (DMS) objects within Chart II. It specifies methods for setting messages and clearing messages from a sign (in maintenance mode), polling a sign, changing the configuration of a sign, and reseting a sign. (Setting messages on a sign in online mode are not accomplished by manipulating a DMS directly; that is accomplished by manipulating traffic events, which interfaces with the DMSArbitrationQueue of a sign. This activity involves the DMS extension, Chart2DMS, which defines interactions with signs under Chart II business rules.)

3.19.1.6.12 DMSArbQueueEntry (Class)

The DMSArbQueueEntry class provides an implementation of ArbQueueEntry that is used for most standard entries placed on the arbitration queue. When its setActive, setInactive, and setFailed methods are called, it adds a log entry to its traffic event and calls the appropriate method on its response plan item (setActive, setInactive, or update).

3.19.1.6.13 DMSConfiguration (Class)

The DMSConfiguration class is an abstract class that describes the configuration of a DMS device. This configuration information is normally fairly static: things like the size of the sign in characters and pixels, its name and location, and how to contact the sign (as opposed to dynamic information like the current message on the sign, which is defined in an analogous Status object).

3.19.1.6.14 DMSConfigurationEventInfo (Class)

The DMSConfigurationEventInfo class is the type of DMSEvent used for DMSEventType DMSConfigChanged. It contains a DMSConfiguration object that details the new configuration for a Chart II DMS object.

3.19.1.6.15 DMSEvent (Class)

The DMSEvent class is a union which can be any one of four events relating to DMS operations which can be pushed on an Event Channel to update event consumers on DMS-related activities. The four types of events, defined by the enumeration DMSEventType, are: DMSAdded, DMSDeleted, CurrentDMSStatus, and DMSConfigChanged.

3.19.1.6.16 DMSEventType (Class)

The DMSEventType is an enumeration which defines the four types of events relating to DMS operations which can be pushed on an Event Channel to update event consumers on DMS-related activities. The four types of events are: DMSAdded, DMSDeleted, CurrentDMSStatus, and DMSConfigChanged.

3.19.1.6.17 DMSFactory (Class)

The DMSFactory class specifies the interface to be used to create DMS objects within the Chart II system. It also provides a method to get a list of DMS devices currently in the system.

3.19.1.6.18 DMSList (Class)

The DMSList class is simply a list of DMS devices which can be used by the DMS Factory and other classes for maintaining the list or other lists of DMS objects.

3.19.1.6.19 DMSMessage (Class)

The DMSMessage class is an abstract class that describes a message for a DMS. It consists of two elements: a MULTI-formatted message and beacon state information (whether the message requires that the beacons be on). The DMSMessage is contained within a DMSStatus object, used to communicate the current message on a sign, and so within a DMSRPIData object, used to specify the message that should be on a sign when the response plan item is executed.

3.19.1.6.20 DMSPlanItemData (Class)

The DMSPlanItemData class is a valuetype that contains data stored in a plan item for a DMS. It is derived from PlanItemData.

3.19.1.6.21 DMSRPIData (Class)

The DMSRPIData class is an abstract class that describes a response plan item for a DMS. It contains the unique identifier of the DMS to contain the DMSMessage, and the DMSMessage itself.

3.19.1.6.22 DMSStatus (Class)

The DMSStatus class is an abstract value-type class that provides status information for a DMS. This status information is relatively dynamic: things like the current message on the sign, its beacon state, its current operational mode (online, offline, maintenance mode), and current operational status (OK, COMM_FAILURE, or HARDWARE_FAILURE). (More static information about the sign, such as its size and location, is defined in an analogous Configuration object.)

3.19.1.6.23 DMSStatusEventInfo (Class)

The DMSStatusEventInfo class is the type of DMSEvent used for DMSEventType CurrentDMSStatus. It contains a DMSStatus object that details the new status for a Chart II DMS object.

3.19.1.6.24 DMSTestType (Class)

The DMSTestType enumeration identifies two types of tests which can be performed on DMS devices: random and permutation.

3.19.1.6.25 FontMetrics (Class)

The FontMetrics class is a non-behavioral class (structure) which contains information regarding to the font size used on a DMS. It is a part of a DMSConfiguration object.

3.19.1.6.26 FP9500Configuration (Class)

The FP9500Configuration class is an abstract class that extends the Chart2DMSConfiguration class to provide configuration information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.19.1.6.27 FP9500DMS (Class)

The FP9500DMS class extends the Chart2DMS interface and defines a more detailed interface to be used in manipulating FP9500 models of DMS signs. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific DMS control. For instance, the FP9500DMS has a performPixelTest method, which knows how to invoke and interpret a pixel test as supported by the FP9500 model DMS.

3.19.1.6.28 FP9500Status (Class)

The FP9500Status class is an abstract class that extends the Chart2DMSStatus class to provide status information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information. In this case, additional information provided the FP9500 model includes the current message number and current message source.

3.19.1.6.29 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.19.1.6.30 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.19.1.6.31 HARNotifierArbQueueEntry (Class)

The HarNotifierArbQueueEntry class provides an implementation of the ArbQueueEntry used for entries that are placed on the arbitration queue to put a "SHAZAM" message on a DMS. These types of messages have a low priority and are not allowed to overwrite any standard message (from a DMSArbQueueEntry) that is currently displayed on a device. These types of messages are also different in that they are not added to the queue directly by a response plan item and are instead included as a sub-task of activating a message on a HAR. The HAR uses a command status object to track the progress of the HAR notifier message.

3.19.1.6.32 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.19.1.6.33 MULTIParseFailure (Class)

The MULTIParseFailure class is an exception to be thrown when a MULTI-formatted DMS message cannot be correctly parsed.

3.19.1.6.34 MULTIString (Class)

The MULTIString class is a MULTI-formatted DMS message. The DMSMessage class contains a MULTIString value to specify the content of the sign, in addition to the beacon state value.

3.19.1.6.35 NetworkConnectionSite (Class)

The NetworkConnectionSite class contains a string that is used to specify where a service is running. This field is useful for administrators in debugging problems should an object become "software comm failed". It is included in the Chart2DMSStatus.

3.19.1.6.36 OperationalStatus (Class)

The OperationalStatus class enumerates the types of operational status a DMS can have: OK (normal mode), COMM_FAILURE (no communications to the device), or HARDWARE_FAILURE (device is reachable but is reporting a hardware failure). The DMSStatus class contains a value of this type.

3.19.1.6.37 PlanItemData (Class)

This class is a valuetype that is the base class for data stored in a plan item. Derived classes contain specific data that map a device to an operation and the data needed for the operation. For example a derived class provides a mapping between a specific DMS and a DMSMessage.

3.19.1.6.38 ResponsePlanItemData (Class)

This class is a delegate used to perform the execute and remove tasks for the response plan item. Derived classes of this base class have specific implementations for the type of device the response plan item is used to control.

3.19.1.6.39 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.19.1.6.40 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn’t have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.19.1.6.41 ShortErrorStatus (Class)

The ShortErrorStatus class identifies an error condition for a DMS. It is a bit field defined by the NTCIP center to field standard for DMS that specifies error conditions that may be present on the device. This class is used to encapsulate the bit mask and provide a user-friendly interface to the error conditions. The DMSStatus class contains a value of this type.

3.19.1.6.42 SignMetrics (Class)

The SignMetrics class is a non-behavioral class (structure) which contains information regarding to the size of a DMS, in pixels and characters. It is a part of a DMSConfiguration object.

3.19.1.6.43 SignType (Class)

The SignType class defines the sign type for a DMS. Its values are defined by the SignTypeValues class. It is a part of a DMSConfiguration object.

3.19.1.6.44 SignTypeValues (Class)

The SignTypeValues class enumerates the various sign types DMS devices. Examples are bos, cms, vmsChar, etc.

3.19.1.6.45 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.7 PlanManagement (Class Diagram)

This class diagram contains the interfaces used in the creation and management of plans. A plan is a group of actions that are set-up in advance to be used in response to a traffic event. Given the unpredictable nature of traffic events, pre-defined plans are usually only useful for congestion, safety messages, and weather-related messages.

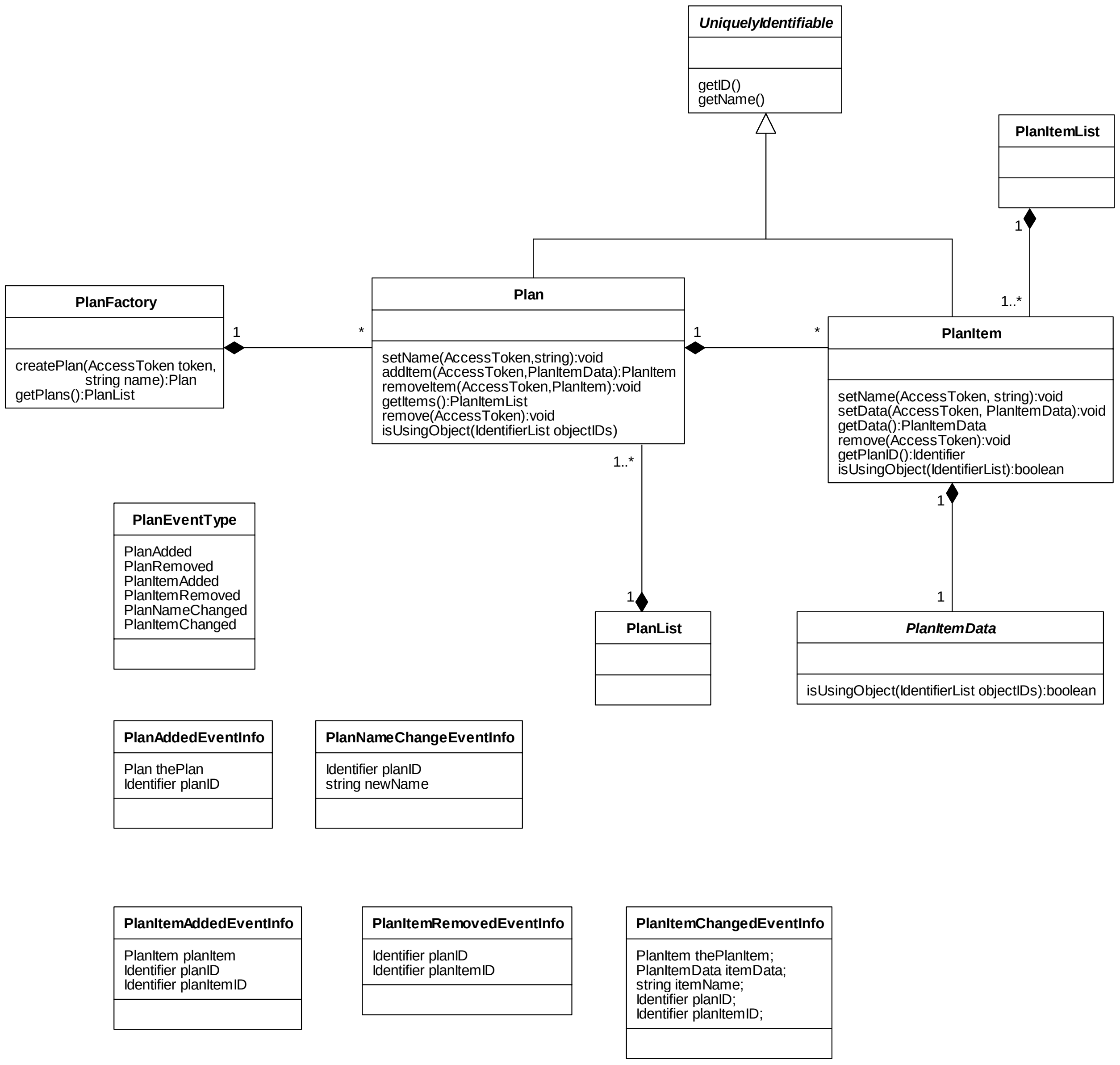


Figure 217. PlanManagement (Class Diagram)

3.19.1.7.1 Plan (Class)

A Plan is a group of actions that are listed out in advance to be used in response to a traffic event. Each action is defined to be a Plan item. The Plan supports functionality to add and remove plan items.

3.19.1.7.2 PlanAddedEventInfo (Class)

The PlanAddedEventInfo class defines the data passed in the PlanAdded event.

3.19.1.7.3 PlanEventType (Class)

The PlanEventType class is an enumeration that describes the types of events that can be pushed for plans. When a plan item is added or modified it is up to the derived item type to push the appropriate type of event.

3.19.1.7.4 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans that can be used in the system.

3.19.1.7.5 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This CORBA interface is subclassed for specific actions that can be planned in the system.

3.19.1.7.6 PlanItemAddedEventInfo (Class)

The PlanItemAddededEventInfo class defines the data passed in the PlanItemAdded event.

3.19.1.7.7 PlanItemChangedEventInfo (Class)

The PlanItemChangedEventInfo class defines the data passed in the PlanItemChanged event.

3.19.1.7.8 PlanItemData (Class)

This class is a valuetype that is the base class for data stored in a plan item. Derived classes contain specific data that map a device to an operation and the data needed for the operation. For example a derived class provides a mapping between a specific DMS and a DMSMessage.

3.19.1.7.9 PlanItemList (Class)

The PlanItemList class is simply a collection of PlanItem objects.

3.19.1.7.10 PlanItemRemovedEventInfo (Class)

The PlanItemRemovedEventInfo defines the data passed in the PlanItemRemoved event.

3.19.1.7.11 PlanList (Class)

The PlanList class is simply a collection of Plan objects.

3.19.1.7.12 PlanNameChangeEventInfo (Class)

The PlanNameChangeEventInfo class defines the data passed in the PlanNameChanged event.

3.19.1.7.13 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.8 HARControl (Class Diagram)

This class diagram contains the interfaces used relating to the control of Highway Advisory Radio (HAR).

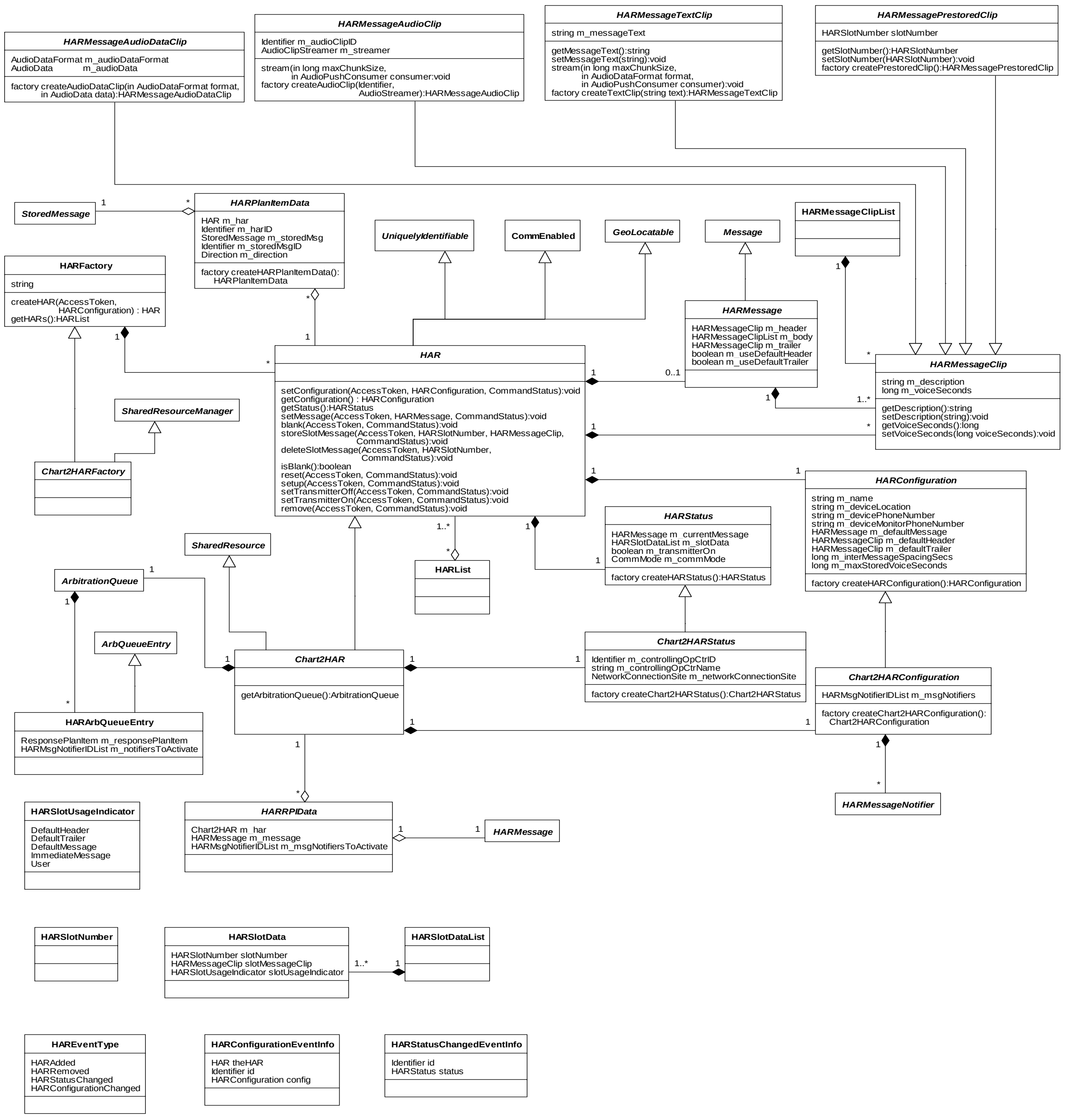

HARMessageAudioDataClip
AudioDataFormat m_audioDataFormat
AudioData m_audioData
factory createAudioDataClip(in AudioDataFormat format, in AudioData data):HARMessageAudioDataClip
HARMessageAudioClip
Identifier m_audioClipID
AudioClipStreamer m_streamer
stream(in long maxChunkSize, in AudioPushConsumer consumer:void
factory createAudioClip(Identifier, AudioStreamer):HARMessageAudioClip
HARMessageTextClip
string m_messageText
getMessageText():string
setMessageText(string):void
stream(in long maxChunkSize, in AudioDataFormat format, in AudioPushConsumer consumer):void
factory createTextClip(string text):HARMessageTextClip
HARMessagePrestoredClip
HARSlotNumber slotNumber
getSlotNumber():HARSlotNumber
setSlotNumber(HARSlotNumber):void
factory createPrestoredClip():HARMessagePrestoredClip
StoredMessage
HARPlanItemData
HAR m_har
Identifier m_harID
StoredMessage m_storedMsg
Identifier m_storedMsgID
Direction m_direction
factory createHARPlanItemData(): HARPlanItemData
UniquelyIdentifiable
CommEnabled
GeoLocatable
Message
HARMessageClipList
HARFactory
string
createHAR(AccessToken, HARConfiguration) : HAR
getHARs():HARList
HARMessage
HARMessageClip m_header
HARMessageClipList m_body
HARMessageClip m_trailer
boolean m_useDefaultHeader
boolean m_useDefaultTrailer
HAR
setConfiguration(AccessToken, HARConfiguration, CommandStatus):void
getConfiguration() : HARConfiguration
getStatus():HARStatus
setMessage(AccessToken, HARMessage, CommandStatus):void
blank(AccessToken, CommandStatus):void
storeSlotMessage(AccessToken, HARSlotNumber, HARMessageClip, CommandStatus):void
deleteSlotMessage(AccessToken, HARSlotNumber, CommandStatus):void
isBlank():boolean
reset(AccessToken, CommandStatus):void
setup(AccessToken, CommandStatus):void
setTransmitterOff(AccessToken, CommandStatus):void
setTransmitterOn(AccessToken, CommandStatus):void
remove(AccessToken, CommandStatus):void
HARMessageClip
string m_description
long m_voiceSeconds
getDescription():string
setDescription(string):void
getVoiceSeconds():long
setVoiceSeconds(long voiceSeconds):void
SharedResourceManager
Chart2HARFactory
HARConfiguration
string m_name
string m_deviceLocation
string m_devicePhoneNumber
string m_deviceMonitorPhoneNumber
HARMessage m_defaultMessage
HARMessageClip m_defaultHeader
HARMessageClip m_defaultTrailer
long m_interMessageSpacingSecs
long m_maxStoredVoiceSeconds
factory createHARConfiguration():HARConfiguration
HARStatus
HARMessage m_currentMessage
HARSlotDataList m_slotData
boolean m_transmitterOn
CommMode m_commMode
factory createHARStatus():HARStatus
SharedResource
HARList
ArbitrationQueue
ArbQueueEntry
Chart2HAR
getArbitrationQueue():ArbitrationQueue
Chart2HARStatus
Identifier m_controllingOpCtrlD
string m_controllingOpCtrName
NetworkConnectionSite m_networkConnectionSite
factory createChart2HARStatus():Chart2HARStatus
Chart2HARConfiguration
HARMsgNotifierIDList m_msgNotifiers
factory createChart2HARConfiguration(): Chart2HARConfiguration
HARArbQueueEntry
ResponsePlanItem m_responsePlanItem
HARMsgNotifierIDList m_notifiersToActivate
HARMessageNotifier
HARSlotUsageIndicator
DefaultHeader
DefaultTrailer
DefaultMessage
ImmediateMessage
User
HARRPIData
Chart2HAR m_har
HARMessage m_message
HARMsgNotifierIDList m_msgNotifiersToActivate
HARMessage
HARSlotNumber
HARSlotData
HARSlotNumber slotNumber
HARMessageClip slotMessageClip
HARSlotUsageIndicator slotUsageIndicator
HARSlotDataList
HAREventType
HARAdded
HARRemoved
HARStatusChanged
HARConfigurationChanged
HARConfigurationEventInfo
HAR theHAR
Identifier id
HARConfiguration config
HARStatusChangedEventInfo
Identifier id
HARStatus status

Figure 218. HARControl (Class Diagram)

3.19.1.8.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue performs no special processing when resumed because the queue cleans itself when interrupted and does not allow new entries while interrupted.

3.19.1.8.2 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.19.1.8.3 Chart2HAR (Class)

The Chart2HAR class is an extension of the HAR that is aware of Chart2 business rules, such as arbitration queues, linking device usage to traffic events, and the concept of a shared resource.

3.19.1.8.4 Chart2HARConfiguration (Class)

This class contains configuration data for the HAR that is used for CHART II specific processing (as opposed to the configuration values contained in HARConfiguration that relate to typical HAR usage).

3.19.1.8.5 Chart2HARFactory (Class)

This interface defines objects capable of creating Chart2HAR objects. This factory is also responsible for monitoring the HARs as shared resources and must report when a HAR that is currently broadcasting a message (other than the default) does not have a user logged into the system that is from the controlling operations center.

3.19.1.8.6 Chart2HARStatus (Class)

This class contains status information for a Chart2HAR object. This information is specific to Chart II processing and extends beyond the status related to typical HAR device control.

3.19.1.8.7 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.19.1.8.8 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.19.1.8.9 HAR (Class)

This class is used to represent a Highway Advisory Radio (HAR) device. A HAR is used to broadcast traffic related information over a localized radio transmitter, making the information available to the traveler.

3.19.1.8.10 HARArbQueueEntry (Class)

This class is an arbitration queue entry used to set the message on a HAR on behalf of a traffic event. This entry also specifies the HARMessageNotifiers to be activated when the message is activated.

3.19.1.8.11 HARConfiguration (Class)

This class contains configuration data for a HAR device.

3.19.1.8.12 HARConfigurationEventInfo (Class)

This class defines data pushed with a HARConfigurationChanged and HARAdded CORBA event.

3.19.1.8.13 HAREventType (Class)

This enumeration defines the types of CORBA events that are pushed on a HARControl event channel.

3.19.1.8.14 HARFactory (Class)

This CORBA interface allows new HAR objects to be added to the system.

3.19.1.8.15 HARList (Class)

The HARList class is simply a collection of HAR objects.

3.19.1.8.16 HARMessage (Class)

This utility class represents a message that is capable of being stored on a HAR. It stores the HAR message as a HAR message header, body and footer. It contains methods to input and output them in different formats.

3.19.1.8.17 HARMessageAudioClip (Class)

This class is a thin wrapper for recorded voice that is to be played on a HAR. This class is passed around the system instead of passing the actual voice data. When the actual voice data is needed to play to the user or to program the HAR device, this object's streamer is used to stream the actual voice data.

3.19.1.8.18 HARMessageAudioDataClip (Class)

This class is a message clip that contains audio data and the format of the audio data. Because audio data can be very large, this type of clip is reserved for use when recorded voice is first entered into the system. Recorded voice that already exists in the system is passed throughout the system using HARMessageAudioClip to avoid sending the large audio data when possible.

3.19.1.8.19 HARMessageClip (Class)

This class represents a section of a HAR message. It can be either plain text that would need to be converted to audio prior to broadcast, or binary format (MP3, WAV, etc.)

3.19.1.8.20 HARMessageClipList (Class)

The HARMessageClipList is a collection of HARMessageClip objects.

3.19.1.8.21 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices which are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.19.1.8.22 HARMessagePrestoredClip (Class)

This class stores data used to identify the usage of a clip that has already been stored on a HAR device.

3.19.1.8.23 HARMessageTextClip (Class)

This class represents a HAR message content object that is in plain text format. This message can be checked for banned words and will be converted into a voice message using a speech engine to broadcast on a HAR device.

3.19.1.8.24 HARPlanItemData (Class)

This class is used to associate a message with a HAR for use in Plans.

3.19.1.8.25 HARRPIData (Class)

This class represents an item in a traffic event response plan that is capable of issuing a command to put a message on a HAR when executed. When the item is executed, it adds the message to the arbitration queue of the specified HAR. When the item is removed from the response plan (manually or implicitly through closing the traffic event) the item asks the HAR’s arbitration queue to remove the message.

3.19.1.8.26 HARSlotData (Class)

This struct defines the data used to identify the contents of a slot in the HAR controller.

3.19.1.8.27 HARSlotDataList (Class)

The HARSlotDataList class is simply a collection of HARSlotData objects.

3.19.1.8.28 HARSlotNumber (Class)

The HARSlotNumber is an integer used to specify slot numbers on a HAR controller.

3.19.1.8.29 HARSlotUsageIndicator (Class)

This enum defines indicators used to show the usage of a given slot in the HAR controller.

3.19.1.8.30 HARStatus (Class)

This class contains data that indicates the current status of a HAR device.

3.19.1.8.31 HARStatusChangedEventInfo (Class)

This class contains data that is pushed when the HARStatusChanged CORBA event is pushed on the HARControl event channel.

3.19.1.8.32 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.19.1.8.33 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.19.1.8.34 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn't have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.19.1.8.35 StoredMessage (Class)

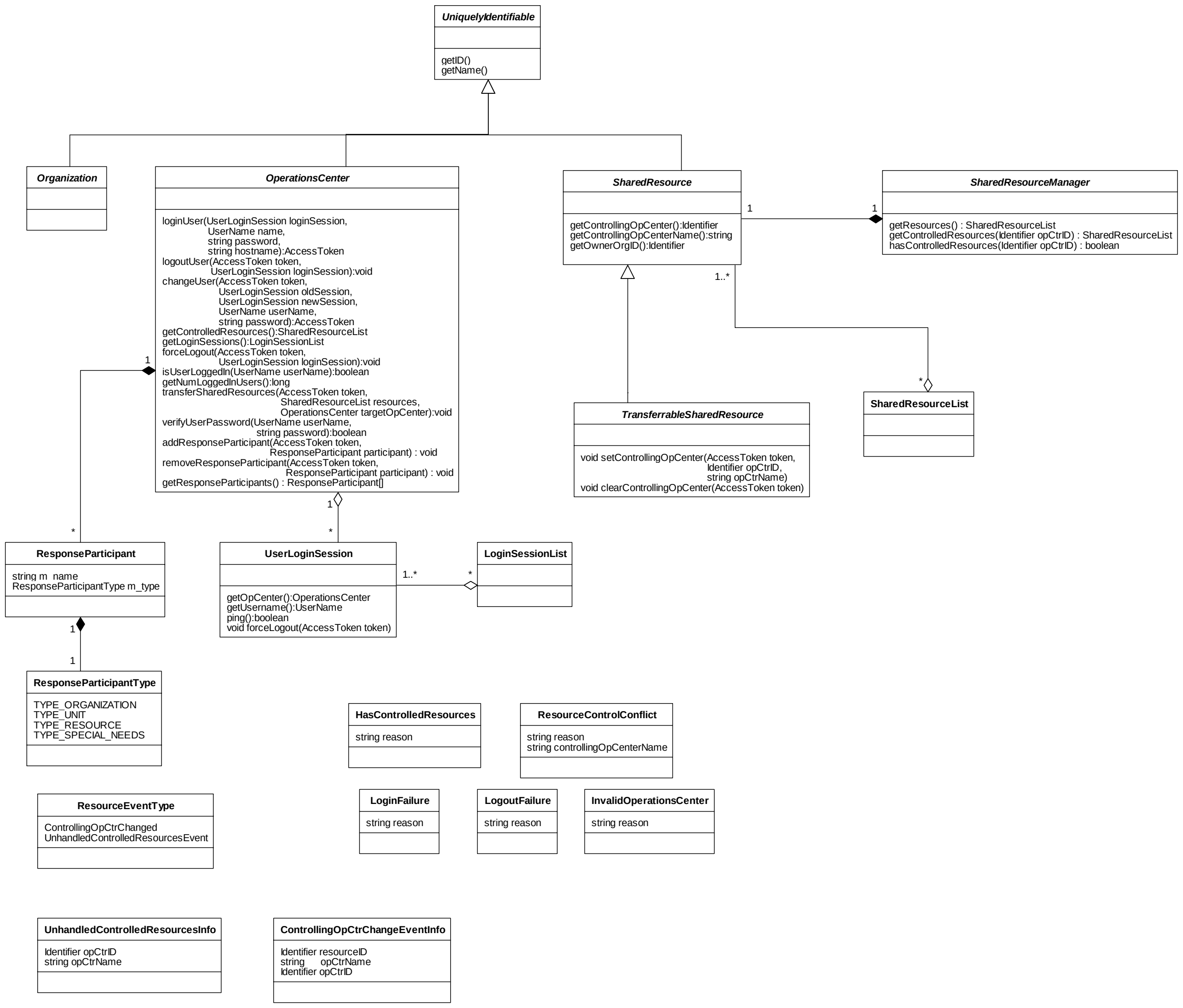
This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description which are used to allow the user to organize messages.

3.19.1.8.36 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.9 ResourceManagement (Class Diagram)

This class diagram contains the interfaces pertaining to shared resources, operations centers, user login sessions, and organizations.



UniquelyIdentifiable
getID()
getName()
Organization
OperationsCenter
loginUser(UserLoginSession loginSession, UserName name, string password, string hostname):AccessToken
logoutUser(AccessToken token, UserLoginSession loginSession):void
changeUser(AccessToken token, UserLoginSession oldSession, UserLoginSession newSession, UserName userName, string password):AccessToken
getControlledResources():SharedResourceList
getLoginSessions():LoginSessionList
forceLogout(AccessToken token, UserLoginSession loginSession):void
isUserLoggedIn(UserName userName):boolean
getNumLoggedInUsers():long
transferSharedResources(AccessToken token, SharedResourceList resources, OperationsCenter targetOpCenter):void
verifyUserPassword(UserName userName, string password):boolean
addResponseParticipant(AccessToken token, ResponseParticipant participant) : void
removeResponseParticipant(AccessToken token, ResponseParticipant participant) : void
getResponseParticipants() : ResponseParticipant[]
SharedResource
getControllingOpCenter():Identifier
getControllingOpCenterName():string
getOwnerOrgID():Identifier
1
1
SharedResourceManager
getResources() : SharedResourceList
getControlledResources(Identifier opCtrlD) : SharedResourceList
hasControlledResources(Identifier opCtrlD) : boolean
1..*
*
SharedResourceList
TransferrableSharedResource
void setControllingOpCenter(AccessToken token, Identifier opCtrlD, string opCtrName)
void clearControllingOpCenter(AccessToken token)
1
*
1
*
ResponseParticipant
string m_name
ResponseParticipantType m_type
UserLoginSession
getOpCenter():OperationsCenter
getUsername():UserName
ping():boolean
void forceLogout(AccessToken token)
1..*
*
LoginSessionList
1
1
ResponseParticipantType
TYPE_ORGANIZATION
TYPE_UNIT
TYPE_RESOURCE
TYPE_SPECIAL_NEEDS
HasControlledResources
string reason
ResourceControlConflict
string reason
string controllingOpCenterName
ResourceEventType
ControllingOpCtrChanged
UnhandledControlledResourcesEvent
LoginFailure
string reason
LogoutFailure
string reason
InvalidOperationsCenter
string reason
UnhandledControlledResourcesInfo
Identifier opCtrlD
string opCtrName
ControllingOpCtrChangeEventInfo
Identifier resourceID
string opCtrName
Identifier opCtrlD

Figure 219. ResourceManagement (Class Diagram)

3.19.1.9.1 ControllingOpCtrChangeEventInfo (Class)

The ControllingOpCtrChangeEventInfo class defines data to be passed on a ControllingOpCtrChange event.

3.19.1.9.2 HasControlledResources (Class)

This class represents an exception which describes a failure caused when the user tries to do something which requires that no resources be controlled, yet the Operations Center which the user is logged in to is still controlling one or more shared resources.

3.19.1.9.3 InvalidOperationsCenter (Class)

Exception which describes a failure caused when the operations center specified is not valid for the attempted operation.

3.19.1.9.4 LoginFailure (Class)

This class represents an exception that describes a login failure.

3.19.1.9.5 LoginSessionList (Class)

A LoginSessionList is simply a collection of UserLoginSession objects.

3.19.1.9.6 LogoutFailure (Class)

This exception is thrown when an error occurs while logging a user out of the system.

3.19.1.9.7 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

3.19.1.9.8 Organization (Class)

The Organization interface extends the UniquelyIdentifiable interface and will represent an organization, that is an administrative body that can control or own resources.

3.19.1.9.9 ResourceControlConflict (Class)

This exception is thrown when attempt to gain control of a shared resource fails because the resource is under the control of a different operations center and the requesting user does not have the functional right to override the restriction.

3.19.1.9.10 ResourceEventType (Class)

The ResourceEventType enumeration defines all of the resource related event types.

3.19.1.9.11 ResponseParticipant (Class)

The ResponseParticipant class is a non-behavioral structure that specifies a participant in a response.

3.19.1.9.12 ResponseParticipantType (Class)

The ResponseParticipantType enumeration defines a type of entity participating in a response to an event. This could be an external organization, a mobile unit, a mobile device or special purpose vehicle, or a special needs vehicle equipped to handle unusual or hazardous situations.

3.19.1.9.13 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.19.1.9.14 SharedResourceList (Class)

A SharedResourceList is simply a collection of SharedResource objects.

3.19.1.9.15 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn’t have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.19.1.9.16 TransferrableSharedResource (Class)

The TransferrableSharedResource interface extends the SharedResource interface, which is implemented by SharedResource objects whose control can be transferred from one operations center to another.

3.19.1.9.17 UnhandledControlledResourcesInfo (Class)

The UnhandledControlledResourcesEvent class is an event pushed when it is detected that an OperationsCenter is controlling one or more controlled resources but has no users logged in.

3.19.1.9.18 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.9.19 UserLoginSession (Class)

The UserLoginSession CORBA interface is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

3.19.1.10 HARNotification (Class Diagram)

This Class Diagram shows the classes involved in manipulating HAR message notifications. The HAR notifiers can be SHAZAMs or DMS devices that are acting as SHAZAMs. Note that R1B2 prevents a DMS SHAZAM message from overwriting another type of DMS message.

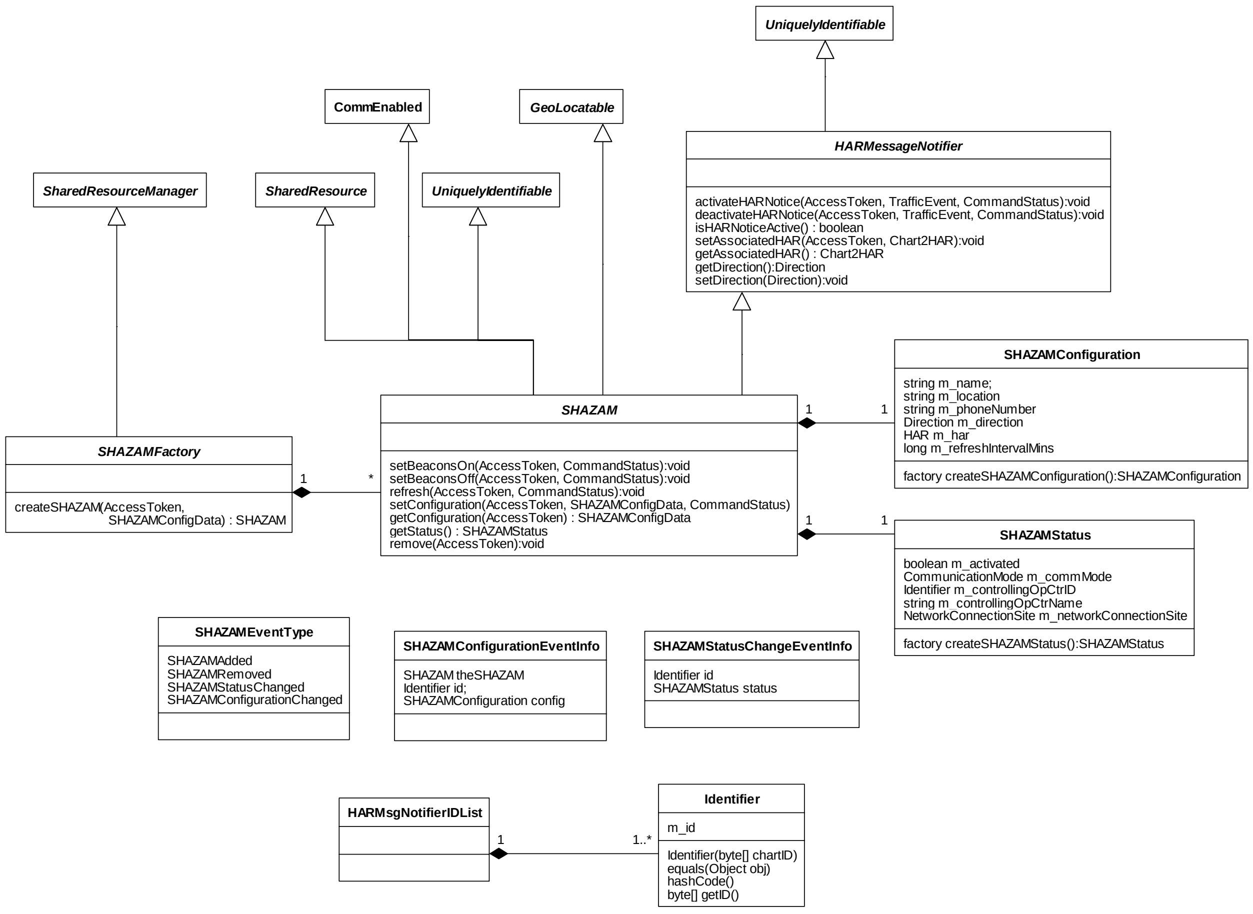

UniquelyIdentifiable
CommEnabled
GeoLocatable
HARMessageNotifier
activateHARNotice(AccessToken, TrafficEvent, CommandStatus):void
deactivateHARNotice(AccessToken, TrafficEvent, CommandStatus):void
isHARNoticeActive() : boolean
setAssociatedHAR(AccessToken, Chart2HAR):void
getAssociatedHAR() : Chart2HAR
getDirection():Direction
setDirection(Direction):void
SharedResourceManager
SharedResource
UniquelyIdentifiable
SHAZAMConfiguration
string m_name;
string m_location
string m_phoneNumber
Direction m_direction
HAR m_har
long m_refreshIntervalMins
factory createSHAZAMConfiguration():SHAZAMConfiguration
SHAZAM
setBeaconsOn(AccessToken, CommandStatus):void
setBeaconsOff(AccessToken, CommandStatus):void
refresh(AccessToken, CommandStatus):void
setConfiguration(AccessToken, SHAZAMConfigData, CommandStatus)
getConfiguration(AccessToken) : SHAZAMConfigData
getStatus() : SHAZAMStatus
remove(AccessToken):void
SHAZAMFactory
createSHAZAM(AccessToken, SHAZAMConfigData) : SHAZAM
1
*
1
1
1
1
SHAZAMStatus
boolean m_activated
CommunicationMode m_commMode
Identifier m_controllingOpCtrID
string m_controllingOpCtrName
NetworkConnectionSite m_networkConnectionSite
factory createSHAZAMStatus():SHAZAMStatus
SHAZAMEventType
SHAZAMAdded
SHAZAMRemoved
SHAZAMStatusChanged
SHAZAMConfigurationChanged
SHAZAMConfigurationEventInfo
SHAZAM theSHAZAM
Identifier id;
SHAZAMConfiguration config
SHAZAMStatusChangeEventInfo
Identifier id
SHAZAMStatus status
HARMsgNotifierIDList
1
1..*
Identifier
m_id
Identifier(byte[] chartID)
equals(Object obj)
hashCode()
byte[] getID()

Figure 220. HARNotification (Class Diagram)

3.19.1.10.1 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.19.1.10.2 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.19.1.10.3 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices which are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.19.1.10.4 HARMsgNotifierIDList (Class)

This typedef is a sequence of HARMessageNotifier identifiers.

3.19.1.10.5 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

3.19.1.10.6 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.19.1.10.7 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn't have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.19.1.10.8 SHAZAM (Class)

This class is used to represent a SHAZAM field device. This class uses a helper class to perform the model specific protocol for device command and control.

3.19.1.10.9 SHAZAMConfiguration (Class)

This class contains data that specifies the configuration of a SHAZAM device.

3.19.1.10.10 SHAZAMConfigurationEventInfo (Class)

This class contains data that is pushed on the SHAZAMControl CORBA event channel with a SHAZAMConfigurationChanged or SHAZAMAdded event type.

3.19.1.10.11 SHAZAMEventType (Class)

This enum defines the types of CORBA events that are pushed on a SHAZAM control event channel.

3.19.1.10.12 SHAZAMFactory (Class)

This CORBA interface allows new SHAZAM objects to be added to the system.

3.19.1.10.13 SHAZAMStatus (Class)

This class contains the current status of a SHAZAM device.

3.19.1.10.14 SHAZAMStatusChangeEventInfo (Class)

This class contains data that is pushed on a SHAZAMControl event channel with a SHAZAMStatusChanged event.

3.19.1.10.15 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.11 LibraryManagement (Class Diagram)

This class diagram shows all classes and relationships relating to message libaries.

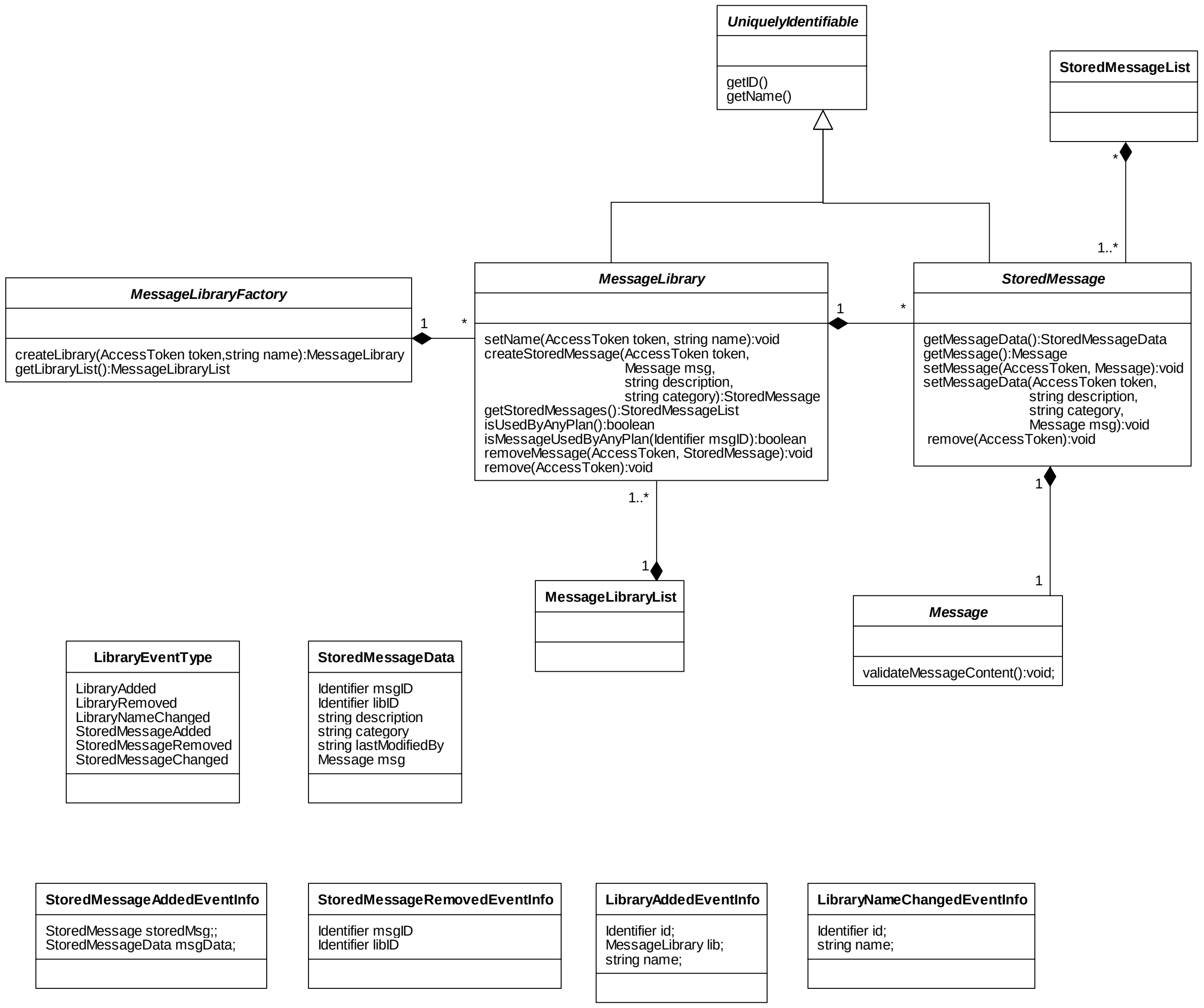

UniquelyIdentifiable
getID()
getName()
StoredMessageList
MessageLibraryFactory
createLibrary(AccessToken token,string name):MessageLibrary
getLibraryList():MessageLibraryList
MessageLibrary
setName(AccessToken token, string name):void
createStoredMessage(AccessToken token, Message msg, string description, string category):StoredMessage
getStoredMessages():StoredMessageList
isUsedByAnyPlan():boolean
isMessageUsedByAnyPlan(Identifier msgID):boolean
removeMessage(AccessToken, StoredMessage):void
remove(AccessToken):void
StoredMessage
getMessageData():StoredMessageData
getMessage():Message
setMessage(AccessToken, Message):void
setMessageData(AccessToken token, string description, string category, Message msg):void
remove(AccessToken):void
MessageLibraryList
Message
validateMessageContent():void;
LibraryEventType
LibraryAdded
LibraryRemoved
LibraryNameChanged
StoredMessageAdded
StoredMessageRemoved
StoredMessageChanged
StoredMessageData
Identifier msgID
Identifier libID
string description
string category
string lastModifiedBy
Message msg
StoredMessageAddedEventInfo
StoredMessage storedMsg;;
StoredMessageData msgData;
StoredMessageRemovedEventInfo
Identifier msgID
Identifier libID
LibraryAddedEventInfo
Identifier id;
MessageLibrary lib;
string name;
LibraryNameChangedEventInfo
Identifier id;
string name;
1
*
1..*

Figure 221. LibraryManagement (Class Diagram)

3.19.1.11.1 LibraryAddedEventInfo (Class)

This struct defines data passed with a DMSLibraryAdded event.

3.19.1.11.2 LibraryEventType (Class)

This enum defines the types of events that can be pushed on a LibraryManagement event channel.

3.19.1.11.3 LibraryNameChangedEventInfo (Class)

This struct defines data passed with a LibraryNameChanged event.

3.19.1.11.4 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.19.1.11.5 MessageLibrary (Class)

This class represents a logical collection of messages that are stored in the database.

3.19.1.11.6 MessageLibraryFactory (Class)

This class is used to create new message libraries and maintain them in a collection.

3.19.1.11.7 MessageLibraryList (Class)

A collection of MessageLibrary objects.

3.19.1.11.8 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description which are used to allow the user to organize messages.

3.19.1.11.9 StoredMessageAddedEventInfo (Class)

This struct defines the data passed with a StoredMessageAdded event.

3.19.1.11.10 StoredMessageData (Class)

This structure defines the data stored in a StoredMessage.

3.19.1.11.11 StoredMessageList (Class)

A collection of StoredMessage objects.

3.19.1.11.12 StoredMessageRemovedEventInfo (Class)

This struct defines data passed with a StoredMessageRemoved event.

3.19.1.11.13 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.19.1.12 LogCommon (Class Diagram)

This class diagram contains all interfaces that are necessary to multiple log types within the CHART II system.

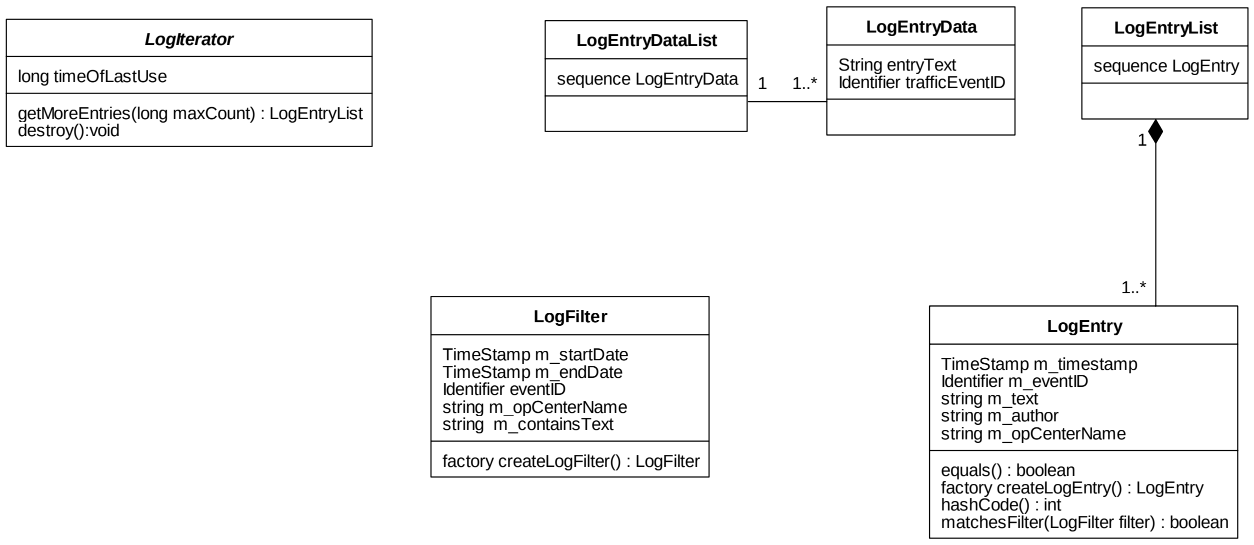


Figure 222. LogCommon (Class Diagram)

3.19.1.12.1 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.19.1.12.2 LogEntryData (Class)

LogEntryData is a collection of data required to create one Log Entry, consisting of text (the body of the event) and an ID that refers to a Traffic Event, if appropriate.

3.19.1.12.3 LogEntryDataList (Class)

The LogEntryDataList is simply a sequence of LogEntryData objects, each of which contain the data needed to create one Log Entry. Normally each LogEntryDataList will contain only one LogEntryData object, but if the CommLog service is unavailable for a time, it is possible that multiple LogEntryData objects may be queued up for insertion into the database.

3.19.1.12.4 LogEntryList (Class)

The LogEntryList is simply a sequence of LogEntry instances returned to a requesting process in one clump. (Some requests return so much data that data is returned in clumps. The initial request returns a LogIterator from which additional LogEntryList sequences can be requested, in order to complete the entire query.

3.19.1.12.5 LogFilter (Class)

This class is used to specify the criteria to be used when getting entries from the Communications Log. The caller would create an object of this type specifying the criteria that each log entry must match in order to be returned.

3.19.1.12.6 LogIterator (Class)

This class represents an iterator to iterate through a collection of log entries. If a retrieval request results in more data than is reasonable to transmit all at once, one clump of entries is returned at first, together with a LogIterator from which additional data can be requested, repeatedly, until all entries are returned or the user cancels the operation.

3.19.1.13 TrafficEventManagement (Class Diagram)

This class diagram contains all classes relating to Traffic Events

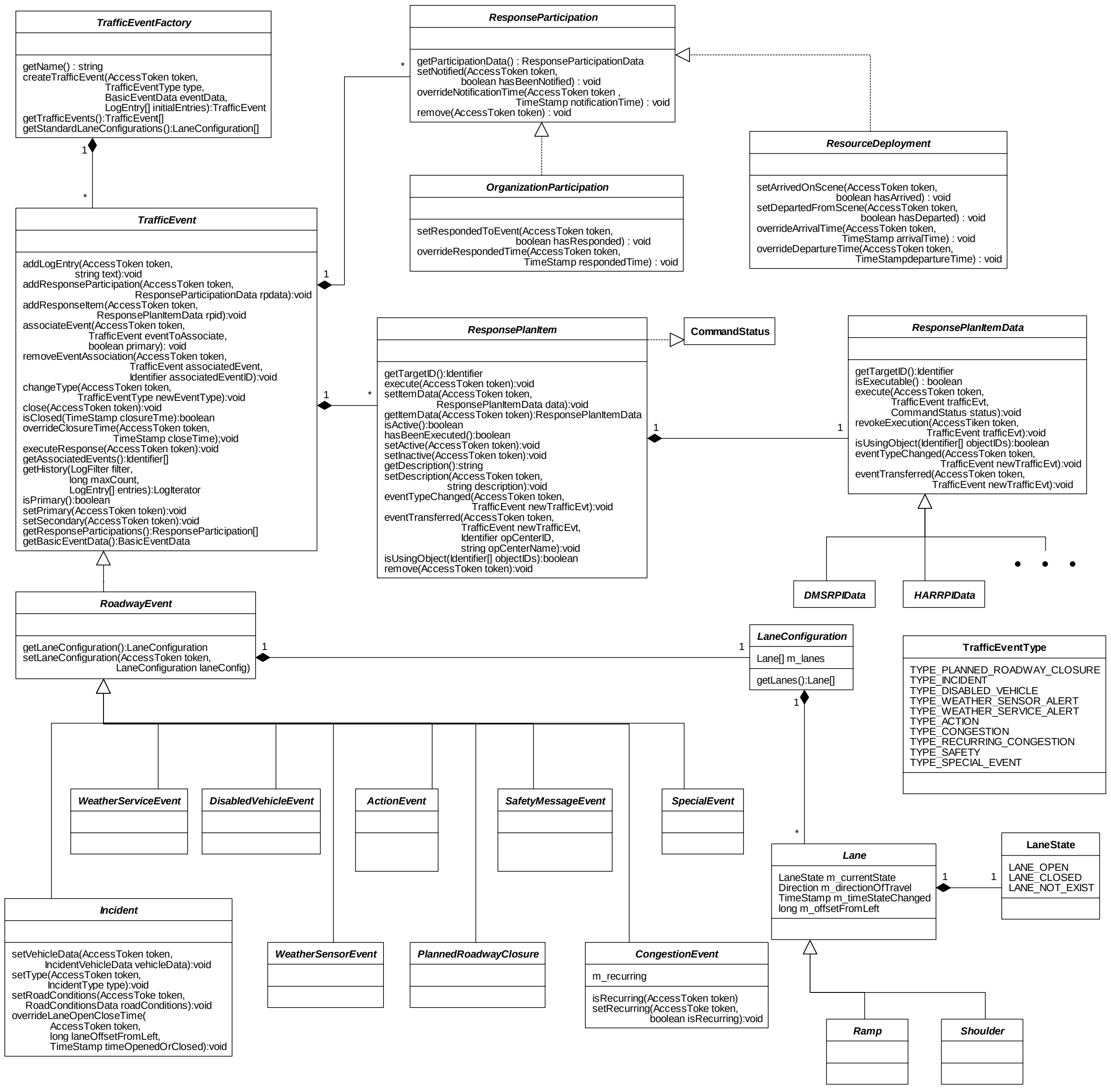

TrafficEventFactory
getName() : string
createTrafficEvent(AccessToken token, TrafficEventType type, BasicEventData eventData, LogEntry[] initialEntries):TrafficEvent
getTrafficEvents():TrafficEvent[]
getStandardLaneConfigurations():LaneConfiguration[]
ResponseParticipation
getParticipationData() : ResponseParticipationData
setNotified(AccessToken token, boolean hasBeenNotified) : void
overrideNotificationTime(AccessToken token , TimeStamp notificationTime) : void
remove(AccessToken token) : void
ResourceDeployment
setArrivedOnScene(AccessToken token, boolean hasArrived) : void
setDepartedFromScene(AccessToken token, boolean hasDeparted) : void
overrideArrivalTime(AccessToken token, TimeStamp arrivalTime) : void
overrideDepartureTime(AccessToken token, TimeStampdepartureTime) : void
OrganizationParticipation
setRespondedToEvent(AccessToken token, boolean hasResponded) : void
overrideRespondedTime(AccessToken token, TimeStamp respondedTime) : void
TrafficEvent
addLogEntry(AccessToken token, string text):void
addResponseParticipation(AccessToken token, ResponseParticipationData rpdata):void
addResponseItem(AccessToken token, ResponsePlanItemData rpid):void
associateEvent(AccessToken token, TrafficEvent eventToAssociate, boolean primary): void
removeEventAssociation(AccessToken token, TrafficEvent associatedEvent, Identifier associatedEventID):void
changeType(AccessToken token, TrafficEventType newEventType):void
close(AccessToken token):void
isClosed(TimeStamp closureTme):boolean
overrideClosureTime(AccessToken token, TimeStamp closeTime);void
executeResponse(AccessToken token):void
getAssociatedEvents():Identifier[]
getHistory(LogFilter filter, long maxCount, LogEntry[] entries):LogIterator
isPrimary():boolean
setPrimary(AccessToken token):void
setSecondary(AccessToken token):void
getResponseParticipations():ResponseParticipation[]
getBasicEventData():BasicEventData
ResponsePlanItem
getTargetID():Identifier
execute(AccessToken token):void
setItemData(AccessToken token, ResponsePlanItemData data):void
getItemData(AccessToken token):ResponsePlanItemData
isActive():boolean
hasBeenExecuted():boolean
setActive(AccessToken token):void
setInactive(AccessToken token):void
getDescription():string
setDescription(AccessToken token, string description):void
eventTypeChanged(AccessToken token, TrafficEvent newTrafficEvt):void
eventTransferred(AccessToken token, TrafficEvent newTrafficEvt, Identifier opCenterID, string opCenterName):void
isUsingObject(Identifier[] objectIDs):boolean
remove(AccessToken token):void
CommandStatus
ResponsePlanItemData
getTargetID():Identifier
isExecutable() : boolean
execute(AccessToken token, TrafficEvent trafficEvt, CommandStatus status):void
revokeExecution(AccessTiken token, TrafficEvent trafficEvt):void
isUsingObject(Identifier[] objectIDs):boolean
eventTypeChanged(AccessToken token, TrafficEvent newTrafficEvt):void
eventTransferred(AccessToken token, TrafficEvent newTrafficEvt):void
DMSRPIData
HARRPIData
RoadwayEvent
getLaneConfiguration():LaneConfiguration
setLaneConfiguration(AccessToken token, LaneConfiguration laneConfig)
LaneConfiguration
Lane[] m_lanes
getLanes():Lane[]
TrafficEventType
TYPE_PLANNED_ROADWAY_CLOSURE
TYPE_INCIDENT
TYPE_DISABLED_VEHICLE
TYPE_WEATHER_SENSOR_ALERT
TYPE_WEATHER_SERVICE_ALERT
TYPE_ACTION
TYPE_CONGESTION
TYPE_RECURRING_CONGESTION
TYPE_SAFETY
TYPE_SPECIAL_EVENT
WeatherServiceEvent
DisabledVehicleEvent
ActionEvent
SafetyMessageEvent
SpecialEvent
Lane
LaneState m_currentState
Direction m_directionOfTravel
TimeStamp m_timeStateChanged
long m_offsetFromLeft
LaneState
LANE_OPEN
LANE_CLOSED
LANE_NOT_EXIST
Incident
setVehicleData(AccessToken token, IncidentVehicleData vehicleData):void
setType(AccessToken token, IncidentType type):void
setRoadConditions(AccessToke token, RoadConditionsData roadConditions):void
overrideLaneOpenCloseTime(AccessToken token, long laneOffsetFromLeft, TimeStamp timeOpenedOrClosed):void
WeatherSensorEvent
PlannedRoadwayClosure
CongestionEvent
m_recurring
isRecurring(AccessToken token)
setRecurring(AccessToke token, boolean isRecurring):void
Ramp
Shoulder
1
*

Figure 223. TrafficEventManagement (Class Diagram)

3.19.1.13.1 ActionEvent (Class)

This class models roadway events that require an operations center to take action but do not fit well into the other event categories. An example of this type of event would be debris in the roadway.

3.19.1.13.2 CommandStatus (Class)

The CommandStatus CORBA interface is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

3.19.1.13.3 CongestionEvent (Class)

This class models roadway congestion that may be tagged as recurring or non-recurring through the use of an attribute.

3.19.1.13.4 DisabledVehicleEvent (Class)

This class models disabled vehicles on the roadway.

3.19.1.13.5 DMSRPIData (Class)

The DMSRPIData class is an abstract class that describes a response plan item for a DMS. It contains the unique identifier of the DMS to contain the DMSMessage, and the DMSMessage itself.

3.19.1.13.6 HARRPIData (Class)

This class represents an item in a traffic event response plan that is capable of issuing a command to put a message on a HAR when executed. When the item is executed, it adds the message to the arbitration queue of the specified HAR. When the item is removed from the response plan (manually or implicitly through closing the traffic event) the item asks the HAR’s arbitration queue to remove the message.

3.19.1.13.7 Incident (Class)

This class models objects representing roadway incidents. An incident typically involves one or more vehicles and roadway lane closures.

3.19.1.13.8 Lane (Class)

This class represents a single traffic lane at the scene of a RoadwayEvent.

3.19.1.13.9 LaneConfiguration (Class)

This class contains data that represents the configuration of the lanes.

3.19.1.13.10 LaneState (Class)

This enumeration lists the possible states that a traffic lane may be in.

3.19.1.13.11 OrganizationParticipation (Class)

This class is used to manage the data captured when an operator notifies another organization of a traffic event.

3.19.1.13.12 PlannedRoadwayClosure (Class)

This class models planned roadway closures such as road construction. This interface will be expanded in future releases to include interfacing with the EORS system.

3.19.1.13.13 Ramp (Class)

This class represents a ramp type traffic lane.

3.19.1.13.14 ResponseParticipation (Class)

This interface represents the involvement of one particular resource or organization in response to a particular traffic event.

3.19.1.13.15 ResponsePlanItem (Class)

Objects of this type can be executed as part of a traffic event response plan. A ResponsePlanItem can be executed by an operator, at which time it becomes the responsibility of the System to activate the item on the ResponseDevice as soon as it is appropriate.

3.19.1.13.16 ResponsePlanItemData (Class)

This class is a delegate used to perform the execute and remove tasks for the response plan item. Derived classes of this base class have specific implementations for the type of device the response plan item is used to control.

3.19.1.13.17 ResourceDeployment (Class)

This class is used to store the data captured when an operator deploys resources to the scene of a traffic event.

3.19.1.13.18 RoadwayEvent (Class)

This class models any type of incident that can occur on a roadway. This point in the heirarchy provides a break off point for traffic event types that pertain to other modals.

3.19.1.13.19 SafetyMessageEvent (Class)

This type of event is created by an operator when he/she would like to send a safety message to a device.

3.19.1.13.20 Shoulder (Class)

This class represents a shoulder type traffic lane.

3.19.1.13.21 SpecialEvent (Class)

This class models special events that affect roadway conditions such as a concert or professional sporting event.

3.19.1.13.22 TrafficEvent (Class)

Objects of this type represent traffic events that require action from system operators.

3.19.1.13.23 TrafficEventFactory (Class)

This interface is supported by objects that are capable of creating traffic event objects in the system.

3.19.1.13.24 TrafficEventType (Class)

This enum defines the types of traffic events that are supported by the system.

3.19.1.13.25 WeatherSensorEvent (Class)

This class models roadway weather events such as snow or fog that are reported by the system’s weather monitoring devices. Operators will need to manually enter the information in these events for this release. In future releases, these events will be automatically generated by the system.

3.19.1.13.26 WeatherServiceEvent (Class)

This class models roadway weather events such as snow or fog that are manually entered by an operator in response to receiving an alert from the national weather service.

3.19.1.14 TrafficEventManagement2 (Class Diagram)

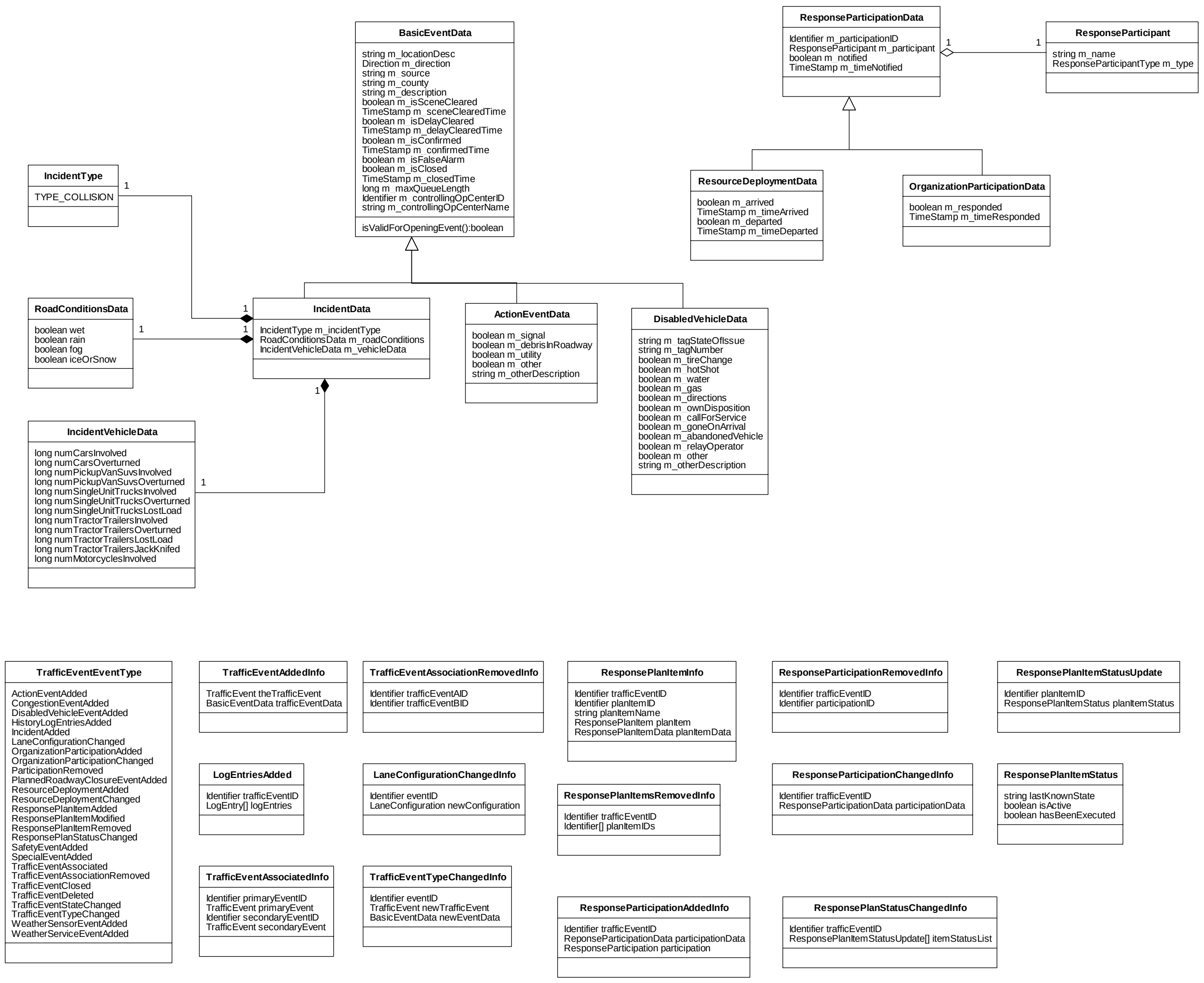

BasicEventData
string m_locationDesc
Direction m_direction
string m_source
string m_county
string m_description
boolean m_isSceneCleared
TimeStamp m_sceneClearedTime
boolean m_isDelayCleared
TimeStamp m_delayClearedTime
boolean m_isConfirmed
TimeStamp m_confirmedTime
boolean m_isFalseAlarm
boolean m_isClosed
TimeStamp m_closedTime
long m_maxQueueLength
Identifier m_controllingOpCenterID
string m_controllingOpCenterName
isValidForOpeningEvent():boolean
ResponseParticipationData
Identifier m_participationID
ResponseParticipant m_participant
boolean m_notified
TimeStamp m_timeNotified
1
1
ResponseParticipant
string m_name
ResponseParticipantType m_type
ResourceDeploymentData
boolean m_arrived
TimeStamp m_timeArrived
boolean m_departed
TimeStamp m_timeDeparted
OrganizationParticipationData
boolean m_responded
TimeStamp m_timeResponded
IncidentType
TYPE_COLLISION
1
RoadConditionsData
boolean wet
boolean rain
boolean fog
boolean iceOrSnow
1
1
1
IncidentData
IncidentType m_incidentType
RoadConditionsData m_roadConditions
IncidentVehicleData m_vehicleData
1
ActionEventData
boolean m_signal
boolean m_debrisInRoadway
boolean m_utility
boolean m_other
string m_otherDescription
DisabledVehicleData
string m_tagStateOfIssue
string m_tagNumber
boolean m_tireChange
boolean m_hotShot
boolean m_water
boolean m_gas
boolean m_directions
boolean m_ownDisposition
boolean m_callForService
boolean m_goneOnArrival
boolean m_abandonedVehicle
boolean m_relayOperator
boolean m_other
string m_otherDescription
IncidentVehicleData
long numCarsInvolved
long numCarsOverturned
long numPickupVanSuvsInvolved
long numPickupVanSuvsOverturned
long numSingleUnitTrucksInvolved
long numSingleUnitTrucksOverturned
long numSingleUnitTrucksLostLoad
long numTractorTrailersInvolved
long numTractorTrailersOverturned
long numTractorTrailersLostLoad
long numTractorTrailersJackKnifed
long numMotorcyclesInvolved
1
TrafficEventEventType
ActionEventAdded
CongestionEventAdded
DisabledVehicleEventAdded
HistoryLogEntriesAdded
IncidentAdded
LaneConfigurationChanged
OrganizationParticipationAdded
OrganizationParticipationChanged
ParticipationRemoved
PlannedRoadwayClosureEventAdded
ResourceDeploymentAdded
ResourceDeploymentChanged
ResponsePlanItemAdded
ResponsePlanItemModified
ResponsePlanItemRemoved
ResponsePlanStatusChanged
SafetyEventAdded
SpecialEventAdded
TrafficEventAssociated
TrafficEventAssociationRemoved
TrafficEventClosed
TrafficEventDeleted
TrafficEventStateChanged
TrafficEventTypeChanged
WeatherSensorEventAdded
WeatherServiceEventAdded
TrafficEventAddedInfo
TrafficEvent theTrafficEvent
BasicEventData trafficEventData
TrafficEventAssociationRemovedInfo
Identifier trafficEventAID
Identifier trafficEventBID
ResponsePlanItemInfo
Identifier trafficEventID
Identifier planItemID
string planItemName
ResponsePlanItem planItem
ResponsePlanItemData planItemData
ResponseParticipationRemovedInfo
Identifier trafficEventID
Identifier participationID
ResponsePlanItemStatusUpdate
Identifier planItemID
ResponsePlanItemStatus planItemStatus
LogEntriesAdded
Identifier trafficEventID
LogEntry[] logEntries
LaneConfigurationChangedInfo
Identifier eventID
LaneConfiguration newConfiguration
ResponsePlanItemsRemovedInfo
Identifier trafficEventID
Identifier[] planItemIDs
ResponseParticipationChangedInfo
Identifier trafficEventID
ResponseParticipationData participationData
ResponsePlanItemStatus
string lastKnownState
boolean isActive
boolean hasBeenExecuted
TrafficEventAssociatedInfo
Identifier primaryEventID
TrafficEvent primaryEvent
Identifier secondaryEventID
TrafficEvent secondaryEvent
TrafficEventTypeChangedInfo
Identifier eventID
TrafficEvent newTrafficEvent
BasicEventData newEventData
ResponseParticipationAddedInfo
Identifier trafficEventID
ReponseParticipationData participationData
ResponseParticipation participation
ResponsePlanStatusChangedInfo
Identifier trafficEventID
ResponsePlanItemStatusUpdate[] itemStatusList

Figure 224. TrafficEventManagement2 (Class Diagram)

3.19.1.14.1 ActionEventData (Class)

This class represents all data specific to an Action event type traffic event.

3.19.1.14.2 BasicEventData (Class)

This class represents the data common to all traffic events. All derived data types will inherit all data shown in this class.

3.19.1.14.3 DisabledVehicleData (Class)

This class represents all data specific to a disabled vehicle traffic event.

3.19.1.14.4 IncidentData (Class)

This class represents data specific to an Incident type traffic event.

3.19.1.14.5 IncidentType (Class)

This enumeration lists all possible incident types.

3.19.1.14.6 IncidentVehicleData (Class)

This class represents the vehicles involved data for incidents. Its purpose is to simplify the exchange of data between GUI and server.

3.19.1.14.7 LaneConfigurationChangedInfo (Class)

This structure contains the data that is broadcast when the lane configuration of a traffic event is changed.

3.19.1.14.8 LogEntriesAdded (Class)

This structure contains the data that is broadcast when new entries are added to the event history log of a traffic event.

3.19.1.14.9 OrganizationParticipationData (Class)

This class represents the data required to describe an organization's participation in the response to a traffic event.

3.19.1.14.10 ResourceDeploymentData (Class)

This class represents the data required to describe a resource's participation in the response to a traffic event.

3.19.1.14.11 ResponseParticipant (Class)

The ResponseParticipant class is a non-behavioral structure which specifies a participant in a response.

3.19.1.14.12 ResponseParticipationData (Class)

This class contains all data pertinent to any class that represents a response participation.

3.19.1.14.13 ResponsePlanItemStatus (Class)

This stucture contains data that describes the current state of a response plan item.

3.19.1.14.14 ResponsePlanStatusChangedInfo (Class)

This structure contains the data that is broadcast when one or more response plan items in the response plan of a traffic event change state.

3.19.1.14.15 RoadConditionsData (Class)

This class represents the data necessary to describe the road conditions at the scene of a traffic event.

3.19.1.14.16 ResponseParticipationAddedInfo (Class)

This structure contains the data that is broadcast when a response participant is added to the response to a particular traffic event.

3.19.1.14.17 ResponseParticipationRemovedInfo (Class)

This structure contains the data that is broadcast when one or more response plan items are removed from a traffic event.

3.19.1.14.18 ResponseParticipationChangedInfo (Class)

This structure contains the data pushed in a CORBA event any time any type of response participation object changes state.

3.19.1.14.19 ResponsePlanItemInfo (Class)

This structure contains the data that is broadcast any time a new response plan item is added or an existing response plan item is modified.

3.19.1.14.20 ResponsePlanItemsRemovedInfo (Class)

This structure contains the data that is broadcast when one or more response plan items are removed from a traffic event.

3.19.1.14.21 ResponsePlanItemStatusUpdate (Class)

This structure contains data that describes a status change to a particular response plan item.

3.19.1.14.22 TrafficEventAddedInfo (Class)

This structure contains the data that is broadcast when a new traffic event is added to the system.

3.19.1.14.23 TrafficEventAssociatedInfo (Class)

This structure contains the data that is broadcast when two traffic events are associated.

3.19.1.14.24 TrafficEventAssociationRemovedInfo (Class)

This structure contains the data that is broadcast when the association between two traffic events is removed.

3.19.1.14.25 TrafficEventEventType (Class)

his enumeration defines the types of CORBA events that can be broadcast on a Traffic Event related CORBA Event channel.

3.19.1.14.26 TrafficEventTypeChangedInfo (Class)

This structure contains the data that is broadcast when a traffic event changes types. The traffic event object that represented the traffic event previously is removed from the system and is replaced by the newTrafficEvent reference contained in this structure. If the consumer of this CORBA event has stored any references to the traffic event previously, those references should be replaced with this new reference.

3.19.1.15 UserManagement (Class Diagram)

This class diagram contains the interfaces necessary to manage and utilize user profiles.

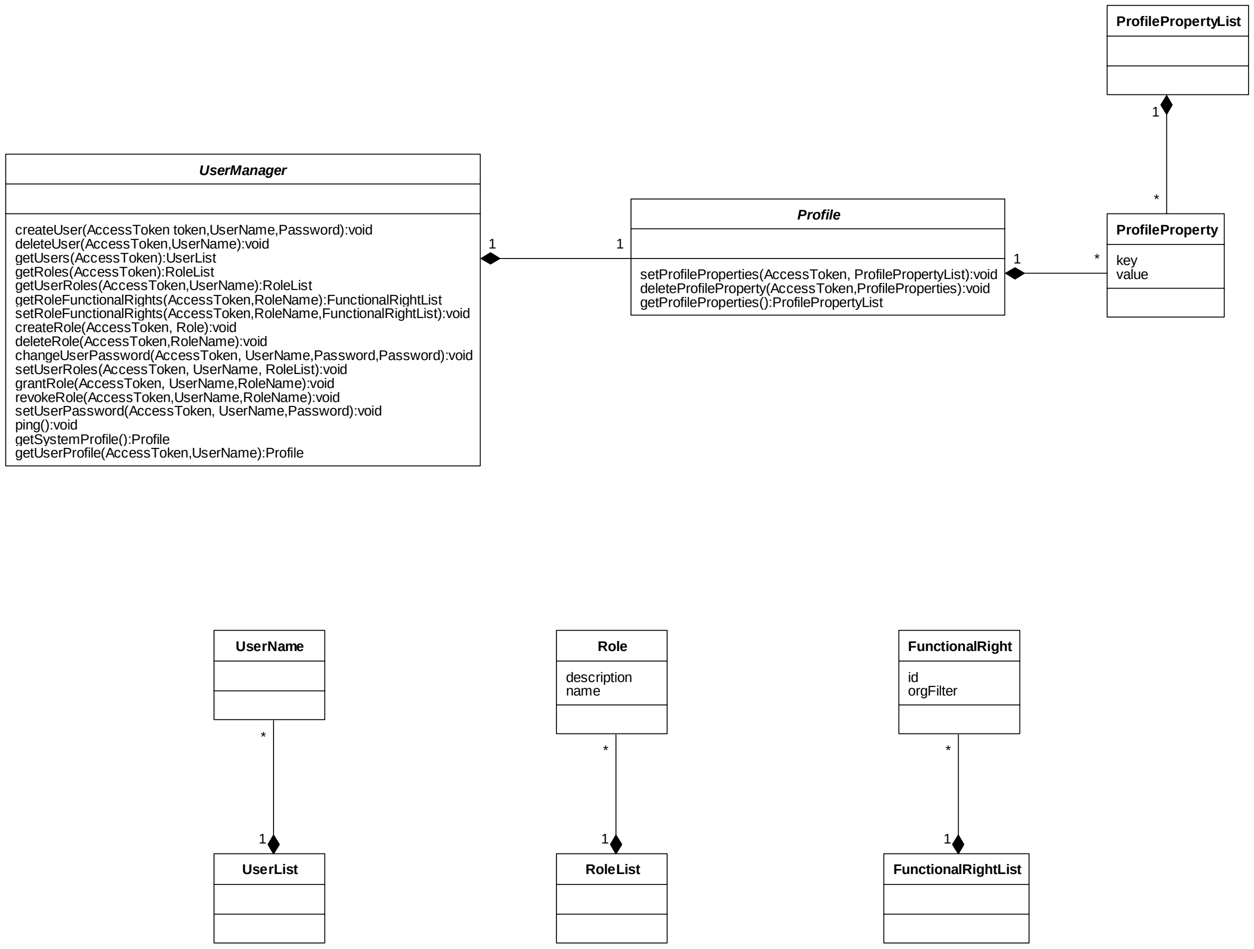

ProfilePropertyList
1
*
UserManager
createUser(AccessToken token,UserName,Password):void
deleteUser(AccessToken,UserName):void
getUsers(AccessToken):UserList
getRoles(AccessToken):RoleList
getUserRoles(AccessToken,UserName):RoleList
getRoleFunctionalRights(AccessToken,RoleName):FunctionalRightList
setRoleFunctionalRights(AccessToken,RoleName,FunctionalRightList):void
createRole(AccessToken, Role):void
deleteRole(AccessToken,RoleName):void
changeUserPassword(AccessToken, UserName,Password,Password):void
setUserRoles(AccessToken, UserName, RoleList):void
grantRole(AccessToken, UserName,RoleName):void
revokeRole(AccessToken,UserName,RoleName):void
setUserPassword(AccessToken, UserName,Password):void
ping():void
getSystemProfile():Profile
getUserProfile(AccessToken,UserName):Profile
1
1
Profile
setProfileProperties(AccessToken, ProfilePropertyList):void
deleteProfileProperty(AccessToken,ProfileProperties):void
getProfileProperties():ProfilePropertyList
1
*
ProfileProperty
key
value
UserName
*
1
UserList
Role
description
name
*
1
RoleList
FunctionalRight
id
orgFilter
*
1
FunctionalRightList

Figure 225. UserManagement (Class Diagram)

3.19.1.15.1 FunctionalRight (Class)

A functional right epresents a particular user capability. A functional right grants a particular capability to perform system functions. Each functional right may be limited by attaching the identifier of a particular organization to which this right is constrained. This capability allows an administrator to grant a particular Role the ability to modify only shared resources owned by the identified organization. The orgFilter identifier CHART2 will allow access to any organizations shared resources.

3.19.1.15.2 FunctionalRightList (Class)

A list of functional rights.

3.19.1.15.3 Profile (Class)

This class contains a set of user or administrator defined properties that are used to configure how the CHART II system behaves or presents information to a user.

3.19.1.15.4 ProfilePropertyList (Class)

A list of profile properties.

3.19.1.15.5 ProfileProperty (Class)

This class represents a key value pair that can be used to store system properties in the system database.

3.19.1.15.6 Role (Class)

A Role is a collection of functional rights. A Role can be granted to a user, thus granting the user all functional rights contained within the role.

3.19.1.15.7 RoleList (Class)

This structure contains a list of roles.

3.19.1.15.8 UserList (Class)

A list of user names.

3.19.1.15.9 UserName (Class)

This typedef defines the type of UserName fields used in system interfaces.

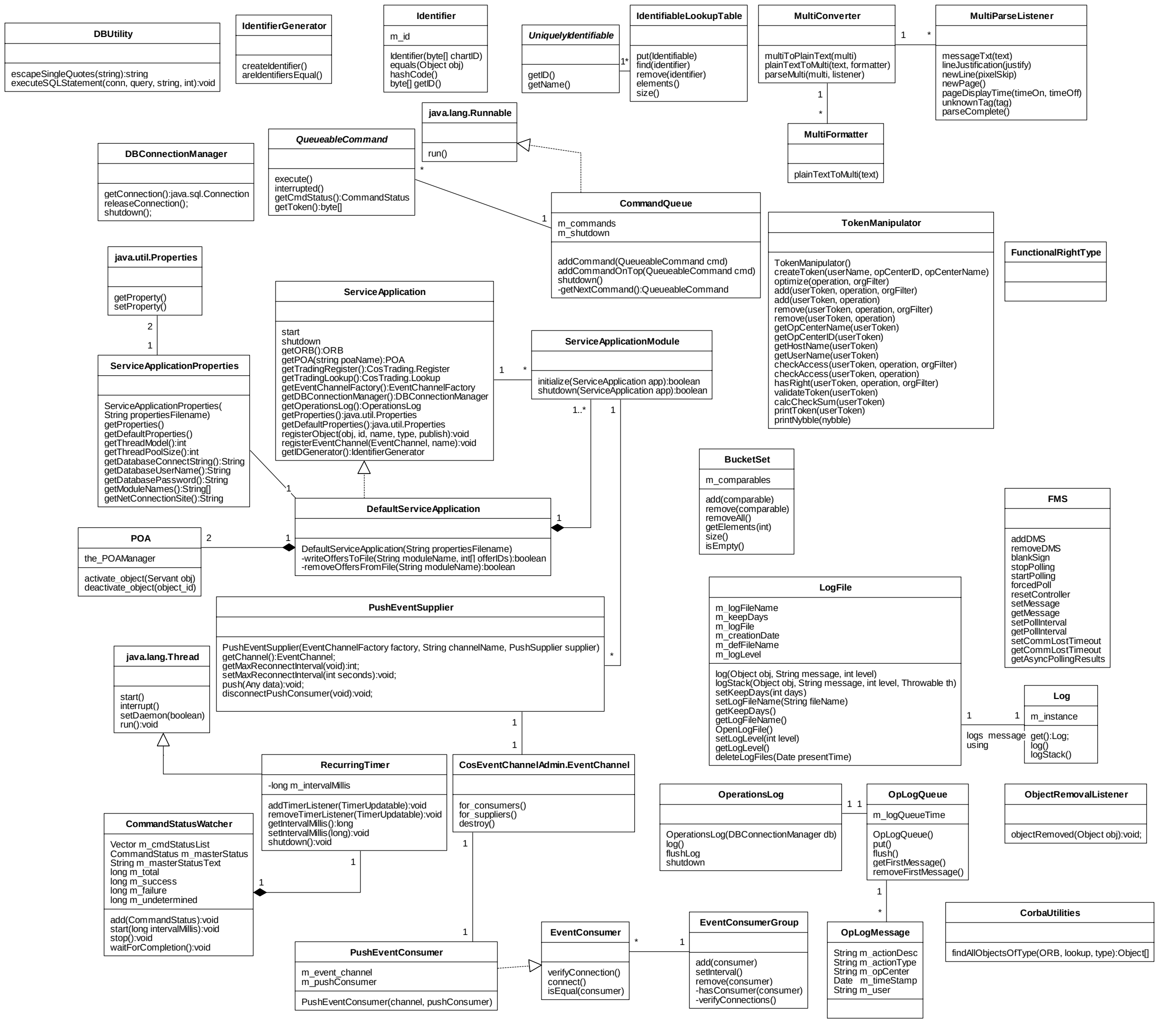
3.19.1.15.10 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

3.20 Utility

3.20.1 Class Diagrams

3.20.1.1 UtilityClasses (Class Diagram)



DBUtility
escapeSingleQuotes(string):string
executeSQLStatement(conn, query, string, int):void
IdentifierGenerator
createIdentifier()
areIdentifiersEqual()
Identifier
m_id
Identifier(byte[] chartID)
equals(Object obj)
hashCode()
byte[] getID()
UniquelyIdentifiable
getID()
getName()
IdentifiableLookupTable
put(Identifiable)
find(identifier)
remove(identifier)
elements()
size()
MultiConverter
multiToPlainText(multi)
plainTextToMulti(text, formatter)
parseMulti(multi, listener)
MultiParseListener
messageTxt(text)
lineJustification(justify)
newLine(pixelSkip)
newPage()
pageDisplayTime(timeOn, timeOff)
unknownTag(tag)
parseComplete()
MultiFormatter
plainTextToMulti(text)
java.lang.Runnable
run()
QueueableCommand
execute()
interrupted()
getCmdStatus():CommandStatus
getToken():byte[]
DBConnectionManager
getConnection():java.sql.Connection
releaseConnection();
shutdown();
CommandQueue
m_commands
m_shutdown
addCommand(QueueableCommand cmd)
addCommandOnTop(QueueableCommand cmd)
shutdown()
-getNextCommand():QueueableCommand
TokenManipulator
TokenManipulator()
createToken(userName, opCenterID, opCenterName)
optimize(operation, orgFilter)
add(userToken, operation, orgFilter)
add(userToken, operation)
remove(userToken, operation, orgFilter)
remove(userToken, operation)
getOpCenterName(userToken)
getOpCenterID(userToken)
getHostName(userToken)
getUserName(userToken)
checkAccess(userToken, operation, orgFilter)
checkAccess(userToken, operation)
hasRight(userToken, operation, orgFilter)
validateToken(userToken)
calcCheckSum(userToken)
printToken(userToken)
printNybble(nybble)
FunctionalRightType
java.util.Properties
getProperty()
setProperty()
ServiceApplication
start
shutdown
getORB():ORB
getPOA(string poaName):POA
getTradingRegister():CosTrading.Register
getTradingLookup():CosTrading.Lookup
getEventChannelFactory():EventChannelFactory
getDBConnectionManager():DBConnectionManager
getOperationsLog():OperationsLog
getProperties():java.util.Properties
getDefaultProperties():java.util.Properties
registerObject(obj, id, name, type, publish):void
registerEventChannel(EventChannel, name):void
getIDGenerator():IdentifierGenerator
ServiceApplicationModule
initialize(ServiceApplication app):boolean
shutdown(ServiceApplication app):boolean
ServiceApplicationProperties
ServiceApplicationProperties(
String propertiesFilename)
getProperties()
getDefaultProperties()
getThreadModel():int
getThreadPoolSize():int
getDatabaseConnectString():String
getDatabaseUserName():String
getDatabasePassword():String
getModuleNames():String[]
getNetConnectionSite():String
DefaultServiceApplication
DefaultServiceApplication(String propertiesFilename)
-writeOffersToFile(String moduleName, int[] offerIDs):boolean
-removeOffersFromFile(String moduleName):boolean
POA
the_POAManager
activate_object(Servant obj)
deactivate_object(object_id)
BucketSet
m_comparables
add(comparable)
remove(comparable)
removeAll()
getElements(int)
size()
isEmpty()
FMS
addDMS
removeDMS
blankSign
stopPolling
startPolling
forcedPoll
resetController
setMessage
getMessage
setPollInterval
getPollInterval
setCommLostTimeout
getCommLostTimeout
getAsyncPollingResults
PushEventSupplier
PushEventSupplier(EventChannelFactory factory, String channelName, PushSupplier supplier)
getChannel():EventChannel;
getMaxReconnectInterval(void):int;
setMaxReconnectInterval(int seconds):void;
push(Any data):void;
disconnectPushConsumer(void):void;
java.lang.Thread
start()
interrupt()
setDaemon(boolean)
run():void
LogFile
m_logFileName
m_keepDays
m_logFile
m_creationDate
m_defFileName
m_logLevel
log(Object obj, String message, int level)
logStack(Object obj, String message, int level, Throwable th)
setKeepDays(int days)
setLogFileName(String fileName)
getKeepDays()
getLogFileName()
OpenLogFile()
setLogLevel(int level)
getLogLevel()
deleteLogFiles(Date presentTime)
Log
m_instance
get():Log;
log()
logStack()
logs message using
RecurringTimer
-long m_intervalMillis
addTimerListener(TimerUpdatable):void
removeTimerListener(TimerUpdatable):void
getIntervalMillis():long
setIntervalMillis(long):void
shutdown():void
CosEventChannelAdmin.EventChannel
for_consumers()
for_suppliers()
destroy()
CommandStatusWatcher
Vector m_cmdStatusList
CommandStatus m_masterStatus
String m_masterStatusText
long m_total
long m_success
long m_failure
long m_undetermined
add(CommandStatus):void
start(long intervalMillis):void
stop():void
waitForCompletion():void
OperationsLog
OperationsLog(DBConnectionManager db)
log()
flushLog
shutdown
OpLogQueue
m_logQueueTime
OpLogQueue()
put()
flush()
getFirstMessage()
removeFirstMessage()
ObjectRemovalListener
objectRemoved(Object obj):void;
CorbaUtilities
findAllObjectsOfType(ORB, lookup, type):Object[]
OpLogMessage
String m_actionDesc
String m_actionType
String m_opCenter
Date m_timeStamp
String m_user
EventConsumerGroup
add(consumer)
setInterval()
remove(consumer)
-hasConsumer(consumer)
-verifyConnections()
EventConsumer
verifyConnection()
connect()
isEqual(consumer)
PushEventConsumer
m_event_channel
m_pushConsumer
PushEventConsumer(channel, pushConsumer)

Figure 226. UtilityClasses (Class Diagram)

3.20.1.1.1 BucketSet (Class)

This class is designed to contain a collection of comparable objects. All of the objects added to this collection must be of the same concrete type. Each element in the collection has an associated counter that tracks how many times this element has been added. It is then possible to get only the elements which have been added to the collection n times where n is a positive integer value. This class is very useful for creating GUI menu's for multiple objects as it allows all objects to insert their menu items and then allows the user to get only those items that all objects inserted.

3.20.1.1.2 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.20.1.1.3 CommandStatusWatcher (Class)

This class is a utility that monitors one or more command status objects for completion. It periodically checks each command status object's completion code and maintains statistics on the number of failures and successes. It provides a blocking method that waits for all command status objects to complete.

3.20.1.1.4 CosEventChannelAdmin.EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

3.20.1.1.5 CorbaUtilities (Class)

This class is a collection of static CORBA utility methods that can be used by both server and GUI for CORBA Trader service transactions.

3.20.1.1.6 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.20.1.1.7 DBUtility (Class)

This class contains methods that allow interaction with the database.

3.20.1.1.8 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.20.1.1.9 EventConsumer (Class)

This interface provides the methods that any EventConsumer object that would like to be managed in an EventConsumerGroup must implement.

3.20.1.1.10 EventConsumerGroup (Class)

This class represents a collection of event consumers that will be monitored to verify that they do not lose their connection to the CORBA event service. The class will periodically ask each consumer to verify its connection to the event channel on which it is dependant to receive events.

3.20.1.1.11 FMS (Class)

This class represents the CHART II system’s interface to the FMS SNMP manager. Most methods included in this class have an associated method in the FMS SNMP Manager DLL provided by the FMS Subsystem. The other methods in this class exist to provide easier interface to the DLL. As an example, this class contains a blankSign method that actually calls setMessage on the FMS Subsystem with the message set to blank and beacons off.

3.20.1.1.12 FunctionalRightType (Class)

This class acts as an enumuration that lists the types of functional rights possible in the CHART2 system. It contains a static member for each possible functional right.

3.20.1.1.13 IdentifiableLookupTable (Class)

This class uses a hash table implementation to store Identifiable objects for fast lookups.

3.20.1.1.14 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

3.20.1.1.15 IdentifierGenerator (Class)

This class is used to create and manipulate identifiers that are to be used in Identifiable objects.

3.20.1.1.16 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.20.1.1.17 java.lang.Thread (Class)

This class represents a java thread of execution.

3.20.1.1.18 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

3.20.1.1.19 Log (Class)

Singleton log object to allow applications to easily create and utilize a LogFile object for system trace messages.

3.20.1.1.20 LogFile (Class)

This class creates a flat file for writing system trace log messages and purges them at user specified interval. The log files created by this class are used for system debugging and maintenance only and are not to be confused with the system operations log that is modeled by the OperationsLog class.

3.20.1.1.21 MultiConverter (Class)

This class provides methods that perform conversions between the DMS MULTI mark-up language and plain text. It also provides a method that will parse a MULTI message and inform a MultiParseListener of elements found in the message.

3.20.1.1.22 MultiFormatter (Class)

This interface must be implemented by classes which convert plain text DMS messages to MULTI formatted messages.

3.20.1.1.23 MultiParseListener (Class)

A MultiParseListener works in conjunction with the MultiConverter to allow an implementing class to be notified as parsing of a MULTI message occurs. An exemplary use of a MultiParseListener would be the MessageView window which will need to have the MULTI message parsed in order to display it as a pixmap.

3.20.1.1.24 ObjectRemovalListener (Class)

This interface is implemented by objects that wish to be notified of objects being removed from the system. This is typically used by objects that store a collection of other objects, such as a factory, to allow them to remove objects from their collection when the object is to be removed from the system.

3.20.1.1.25 OperationsLog (Class)

This class provides the functionality to add a log entry to the Chart II operations log. At the time of instantiation of this class, it creates a queue for log entries. When a user of this class provides a message to be logged, it creates a time-stamped OpLogMessage object and adds this object to the OpLogQueue. Once queued, the messages are written to the database by the queue driver thread in the order they were queued.

3.20.1.1.26 OpLogQueue (Class)

This class is a queue for messages that are to be put into the system's Operations Log. Messages added to the queue can be removed in FIFO order.

3.20.1.1.27 OpLogMessage (Class)

This class holds data for a message to be stored in the system's Operations Log.

3.20.1.1.28 POA (Class)

This interface represents the portable object adapter used to activate and deactivate servant objects.

3.20.1.1.29 PushEventConsumer (Class)

This class is a utility class that will be responsible for connecting a consumer implementation to an event channel, and maintaining that connection. When the verifyConnection method is called, this object will determine if the channel has been lost and will attempt to re-connect to the channel if it has.

3.20.1.1.30 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled indepently of a supplier's push rate.

3.20.1.1.31 QueueableCommand (Class)

A QueueableCommand is an interface used to represent a command that can be placed on a CommandQueue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed. This interface must be implemented by any device command in order that it may be queued on a CommandQueue. The CommandQueue driver calls the execute method to execute a command in the queue and a call to the interuppted method is made when a CommandQueue is shut down.

3.20.1.1.32 RecurringTimer (Class)

A recurring timer is a thread that notifies each TimerUpdatable object that has been registered on a specified period.

3.20.1.1.33 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.20.1.1.34 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.20.1.1.35 ServiceApplicationProperties (Class)

This class provides methods that allow the DefaultServiceApplication to access the necessary properties from the java properties configuration file. It also provides a default properties file which can be retrieved by anyone holding a ServiceApplication interface reference. This gives each installed service module the opportunity to load default values before retrieving property values from the properties file.

3.20.1.1.36 TokenManipulator (Class)

This class contains all functionality required for user rights in the system. It is the only code in the system that knows how to create, modify and check a user's functional rights. It encapsulates the contents of an octet sequence that will be passed to every secure method. Secure methods should call the checkAccess method to validate the user. Client processes should use the check access method to verify access and optimize to reduce reduce the size of the sequence to only those rights which are necessary to invoke the secure method. The token contains the following information. Token version, Token ID, Token Time Stamp, Username, Op Center ID, Op Center IOR, functional rights

3.20.1.1.37 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

1.1.1.1 UtilityClasses2 (Class Diagram)

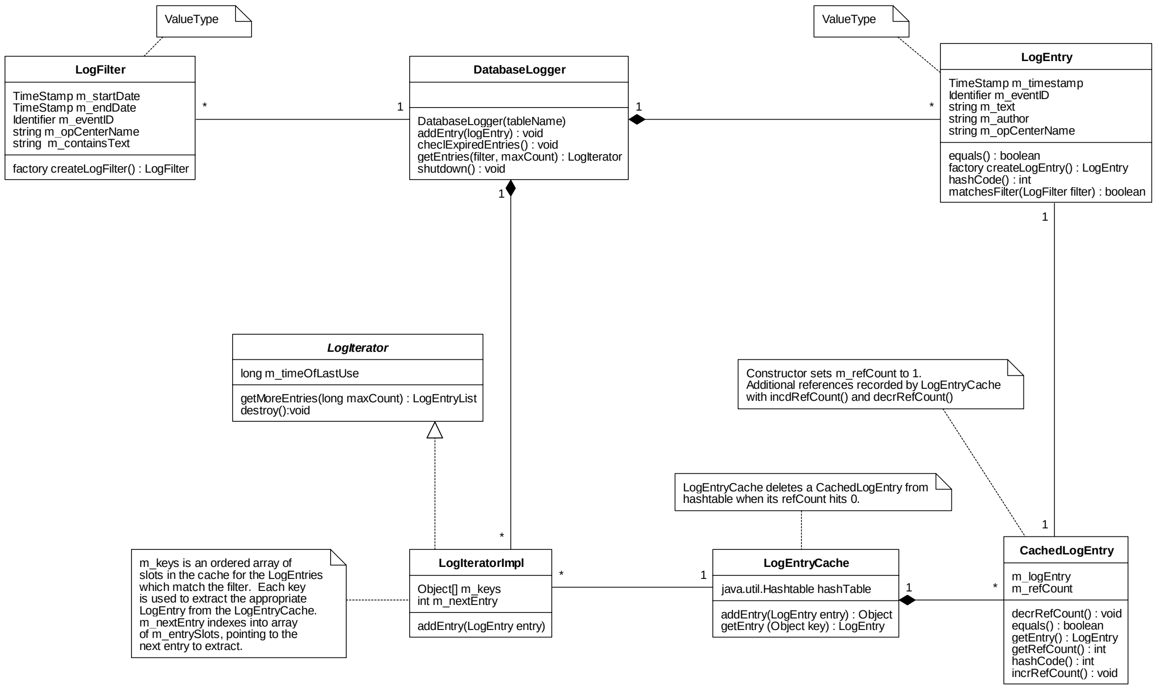

ValueType
LogFilter
TimeStamp m_startDate
TimeStamp m_endDate
Identifier m_eventID
string m_opCenterName
string m_containsText
factory createLogFilter() : LogFilter
ValueType
DatabaseLogger
DatabaseLogger(tableName)
addEntry(logEntry) : void
checlExpiredEntries() : void
getEntries(filter, maxCount) : LogIterator
shutdown() : void
LogEntry
TimeStamp m_timestamp
Identifier m_eventID
string m_text
string m_author
string m_opCenterName
equals() : boolean
factory createLogEntry() : LogEntry
hashCode() : int
matchesFilter(LogFilter filter) : boolean
LogIterator
long m_timeOfLastUse
getMoreEntries(long maxCount) : LogEntryList
destroy():void
Constructor sets m_refCount to 1.
Additional references recorded by LogEntryCache
with incdRefCount() and decrRefCount()
LogEntryCache deletes a CachedLogEntry from
hashtable when its refCount hits 0.
m_keys is an ordered array of
slots in the cache for the LogEntries
which match the filter. Each key
is used to extract the appropriate
LogEntry from the LogEntryCache.
m_nextEntry indexes into array
of m_entrySlots, pointing to the
next entry to extract.
LogIteratorImpl
Object[] m_keys
int m_nextEntry
addEntry(LogEntry entry)
LogEntryCache
java.util.Hashtable hashTable
addEntry(LogEntry entry) : Object
getEntry (Object key) : LogEntry
CachedLogEntry
m_logEntry
m_refCount
decrRefCount() : void
equals() : boolean
getEntry() : LogEntry
getRefCount() : int
hashCode() : int
incrRefCount() : void
*
1
1
*
1
1
*
*
1
1
*
1

Figure 227. UtilityClasses2 (Class Diagram)

3.20.1.1.38 CachedLogEntry (Class)

This class represents a reference-counting object stored in a memory-efficient LogEntryCache. The object of this class encapsulates the stored log entry and adds a reference count.

3.20.1.1.39 DatabaseLogger (Class)

This class represents a generic database logger that can be used to log and retrieve information from the database. This class also provides a mechanism for the user to filter and retrieve logs that meet specific criteria.

3.20.1.1.40 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.20.1.1.41 LogEntryCache (Class)

The LogEntryCache caches log entries returned from a database query which are in excess of the requestor-specified maximum number of entries to return at one time. The LogIterator stores references to the LogEntry objects thus cached, and requests additional objects as needed. The LogEntryCache uses reference counting to prevent storing duplicate copies of LogEntry objects, and it deletes LogEntry objects when they are no longer needed.

3.20.1.1.42 LogFilter (Class)

This class is used to specify the criteria to be used when getting entries from the Communications Log. The caller would create an object of this type specifying the criteria that each log entry must match in order to be returned.

3.20.1.1.43 LogIterator (Class)

This class represents an iterator to iterate through a collection of log entries. If a retrieval request results in more data than is reasonable to transmit all at once, one clump of entries is returned at first, together with a LogIterator from which additional data can be requested, repeatedly, until all entries are returned or the user cancels the operation.

3.20.1.1.44 LogIteratorImpl (Class)

The LogIteratorImpl implements the LogIterator interface; that is, it does the actual work which clients can request via the LogIterator interface. The LogIteratorImpl stores data relating to cached LogEvents for a single retrieval request, and implements the client request to get additional clumps of data pertaining to that request.

3.20.2 Sequence Diagrams

3.20.2.1 DatabaseLogger:getEntries (Sequence Diagram)

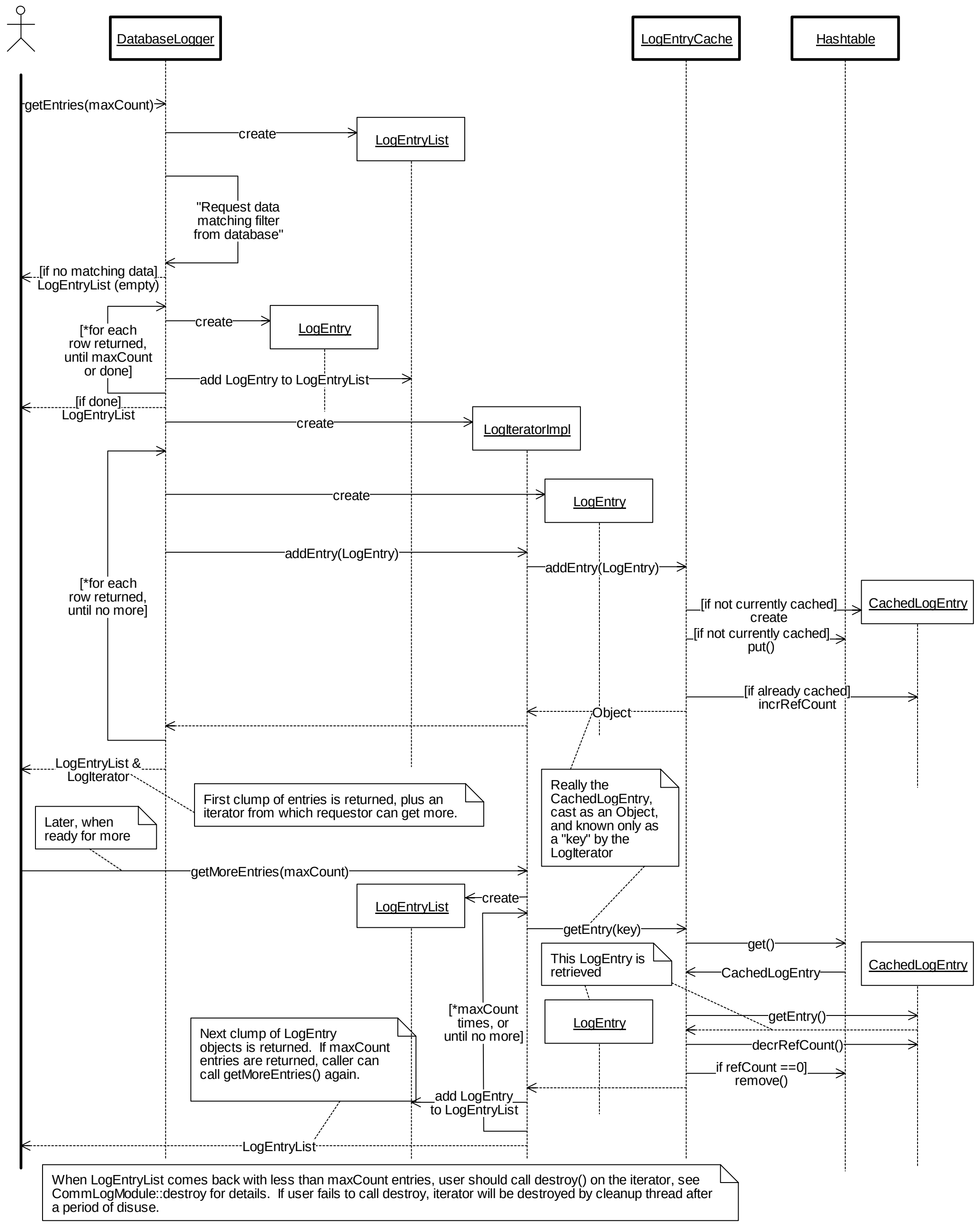

DatabaseLogger
LogEntryCache
Hashtable
getEntries(maxCount)
create
LogEntryList
"Request data matching filter from database"
[if no matching data] LogEntryList (empty)
create
LogEntry
[*for each row returned, until maxCount or done]
add LogEntry to LogEntryList
[if done] LogEntryList
create
LogIteratorImpl
create
LogEntry
addEntry(LogEntry)
addEntry(LogEntry)
[*for each row returned, until no more]
[if not currently cached] create
CachedLogEntry
[if not currently cached] put()
[if already cached] incrRefCount
Object
LogEntryList & LogIterator
First clump of entries is returned, plus an iterator from which requestor can get more.
Really the CachedLogEntry, cast as an Object, and known only as a "key" by the LogIterator
Later, when ready for more
getMoreEntries(maxCount)
LogEntryList
create
getEntry(key)
get()
CachedLogEntry
This LogEntry is retrieved
CachedLogEntry
LogEntry
getEntry()
[*maxCount times, or until no more]
decrRefCount()
Next clump of LogEntry objects is returned. If maxCount entries are returned, caller can call getMoreEntries() again.
if refCount ==0] remove()
add LogEntry to LogEntryList
LogEntryList
When LogEntryList comes back with less than maxCount entries, user should call destroy() on the iterator, see CommLogModule::destroy for details. If user fails to call destroy, iterator will be destroyed by cleanup thread after a period of disuse.

Figure 228. DatabaseLogger:getEntries (Sequence Diagram)

3.20.2.2 DictionaryWrapper:checkForBannedWords (Sequence Diagram)

This diagram shows processing performed by the DictionaryWrapper that is representative of all methods that it duplicates in the Dictionary interface. When a method is called that is to be delegated to a system dictionary, the DictionaryWrapper first attempts to use the dictionary references (if any) that it has already discovered during a previous method invocation. If no references exist (this is true for the first usage of the wrapper) or if all existing references return CORBA failures when used, the DictionaryWrapper queries the trader for all Dictionaries in the system and then attempts to use each until a "live" reference is found or all of the newly discovered references return CORBA failures when used.

A timestamp is used to prevent a flurry of trader queries when no Dictionary objects are available. Prior to doing a trader query to (re)discover dictionaries, the DictionaryWrapper makes sure that at least a minimum amount of time has elapsed since the last time it tried to find a dictionary. The use of synchronization around the discovery process also helps to prevent a flood of trader queries.

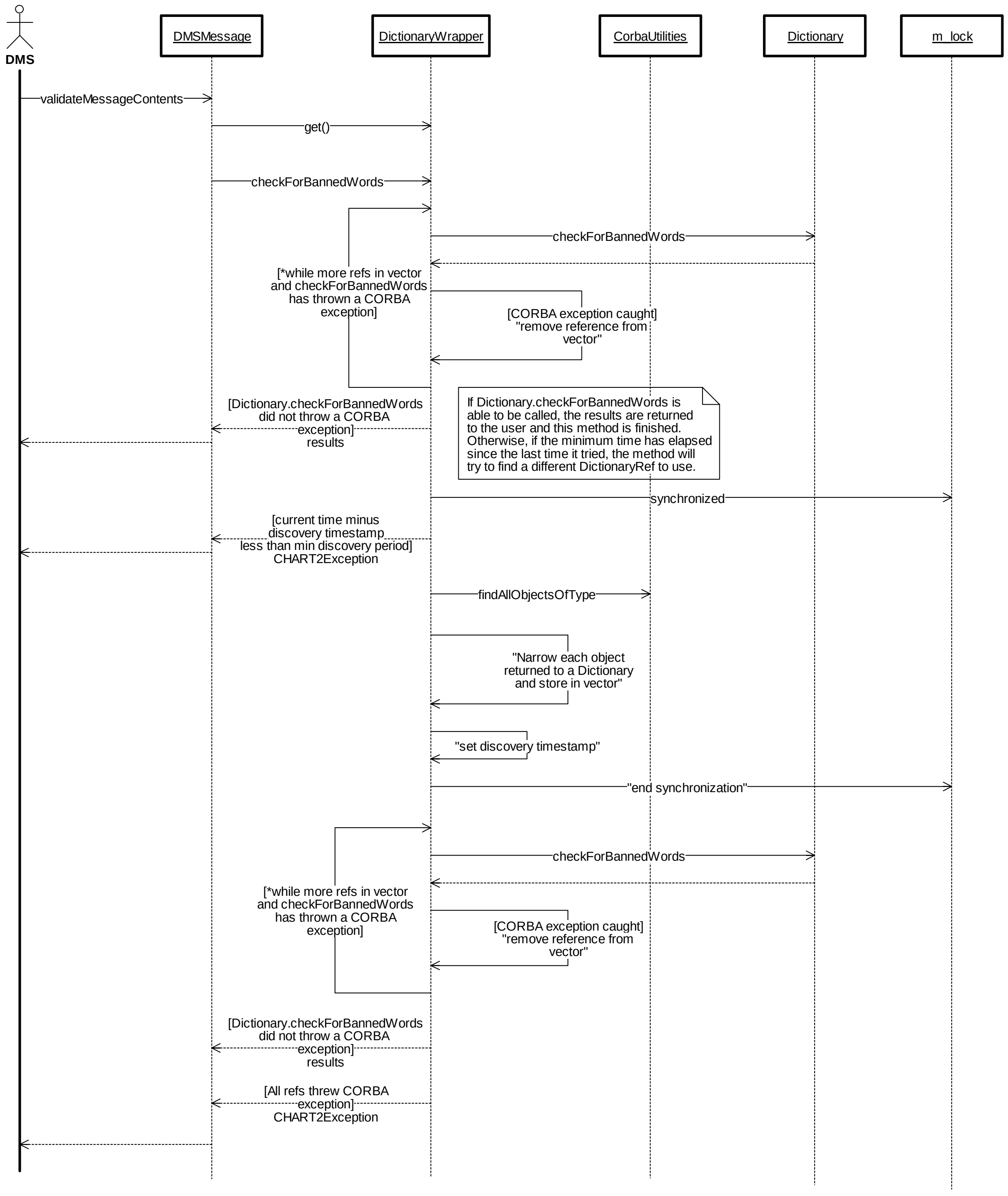

DMS
DMSMessage
DictionaryWrapper
CorbaUtilities
Dictionary
m_lock
validateMessageContents
get()
checkForBannedWords
checkForBannedWords
[*while more refs in vector and checkForBannedWords has thrown a CORBA exception]
[CORBA exception caught] "remove reference from vector"
[Dictionary.checkForBannedWords did not throw a CORBA exception] results
If Dictionary.checkForBannedWords is able to be called, the results are returned to the user and this method is finished. Otherwise, if the minimum time has elapsed since the last time it tried, the method will try to find a different DictionaryRef to use.
synchronized
[current time minus discovery timestamp less than min discovery period] CHART2Exception
findAllObjectsOfType
"Narrow each object returned to a Dictionary and store in vector"
"set discovery timestamp"
"end synchronization"
checkForBannedWords
[*while more refs in vector and checkForBannedWords has thrown a CORBA exception]
[CORBA exception caught] "remove reference from vector"
[Dictionary.checkForBannedWords did not throw a CORBA exception] results
[All refs threw CORBA exception] CHART2Exception

Figure 229. DictionaryWrapper:checkForBannedWords (Sequence Diagram)

Acronyms

The following acronyms appear throughout this document:

API	Application Program Interface
BAA	Business Area Architecture
CORBA	Common Object Request Broker Architecture
DMS	Dynamic Message Sign
EORS	Emergency Operations Reporting System
FMS	Field Management Station
GUI	Graphical User Interface
IDL	Interface Definition Language
ITS	Intelligent Transportation Systems
NTCIP	National Transportation Communications for ITS Protocol
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
R1B2	Release 1, Build 2 of the CHART II System
TTS	Text To Speech
UML	Unified Modeling Language

References

CHART II GUI High Level Design For Release 1 Build 1, document number M361-DS-003R0, Computer Sciences Corporation and PB Farradyne, Inc.

CHART II Release 4 Interim BAA Report, document number M361-BA-004R0, Computer Sciences Corporation and PB Farradyne.

CHART II System Requirements Specification Release 1 Build 2, document number M361-RS-002R1, Computer Sciences Corporation and PB Farradyne.

R1B2 High Level Design, document number M362-DS-005R0, Computer Sciences Corporation and PB Farradyne.

FMS R1B1 High Level Design, document number M303-DS-001R0, Computer Sciences Corporation and PB Farradyne.

The Common Object Request Broker: Architecture and Specification, Revision 2.3.1, OMG Document 99-10-07

Martin Fowler and Kendall Scott, *UML Distilled,* Addison-Wesley, 1997

Appendix A – Glossary

Action Event	A Traffic Event related to the disposition of actions in response to device failures and non-blockage events (e.g. signals, debris, utility, and signs).
Approved Word	A word that is known to the system and has been approved for use when communicating with the motoring public via a messaging device. The dictionary will suggest words to the operator when it encounters a word that has not been previously approved.
Arbitration Queue	A prioritized queue containing messages for display or broadcast on a traveler information device.
Banned Word	A word that may not be used when communicating with the motoring public via a messaging device such as a HAR or DMS.
Comm Log	A collection of information received from any source that requires no action.
Congestion Event	A Traffic Event related to roadway congestion situations. Congestion Events may be recurring or non-recurring.
CORBA Event	A CORBA mechanism using which different Chart2 components exchange information without explicitly knowing about each other.
CORBA Trader	A CORBA service that facilitates object location and discovery. A server advertises an object in the Trading Service based on the kind of service provided by the object. A client locates objects of interest by asking the Trading Service to find all objects that provide a particular service.
Data Model	An object repository that keeps track of changes to the various objects in the repository and informs about these changes as they occur, to observers who are interested in the objects in the repository. A Data Model identifies the subject in a Subject/Observer design pattern.
Dictionary	A collection of banned and approved words.
Deployable Resource	Any resource that can be deployed to the scene in order to provide assistance during a traffic event.
DMS	A Dynamic Message Sign that can be controlled by one Operations Center at a time.
DMS Stored Message Item	A plan item that is used to set a specific message on a specific DMS when added to a Traffic Event response plan and activated.
Emergency Operations Reporting System	A system external to CHART II that (among other things) keeps track of planned roadway closures and permits.

Factory	A CORBA object that is capable of creating other CORBA objects of a particular type. The newly created object will be served from the same process as the factory object that creates it.
FMS	Field Management Station through which the CHART II system communicates with the devices in the field.
Functional Right	A privilege that gives a user the right to perform a particular system action or related group of actions. A functional right may be limited to pertain only to those shared resources owned by a particular organization or can pertain to the shared resources of all organizations.
Graphical User Interface	Part of a software application that provides a graphical interface to its user.
GUI Wrapper Object	A GUI wrapper object is one that wraps a server object to provide it with GUI functionality such as menu handling. It also helps in performance enhancement by caching data locally thereby avoiding network calls when not necessary.
HAR	A Highway Advisory Radio which can be controlled by one Operations Center at a time.
HAR Message	A message which is capable of being stored on a HAR. It is composed of a message header, body and footer.
HAR Message Clip	A message clip is part of a HAR message that could be a header or body or footer. It can be stored either as a text or in one of the binary forms (WAV, MP3 etc).
HAR Message Slot	A message slot is one of the numbered message stores inside the HAR device that can be used to store pre-fabricated messages useful for quick retrieval and playing.
Incident Event	A Traffic Event that is entered by an Operator in response to one of the following types of incidents: Disabled in roadway, Personal injury, Property damage, Fatality, Debris in roadway, Vehicle fire, Maintenance, Signal call, Police activities, Off-road activity, Declaration of emergency, Weather, or Other.
Installable Module	A plugable GUI module that provides a specific function, which when registered with the GUI is called on to initialize itself at the time of GUI startup and shut down at the time of GUI shut down.
Lane Closure	The closure of one or more roadway lanes resulting from a Traffic Event.
Message Library	A collection of stored messages that can be displayed on the DMS or broadcast on a HAR.

Navigator	A Navigator is a GUI window that contains a tree on the left-hand side and a list on the right hand side. Tree elements represent groups of objects and the list on the right hand side represents the objects in the selected group.
Object Discovery	A GUI mechanism in which the client periodically asks the CORBA Trading Service to find objects of those types that are of interest to the GUI, such as DMS, HAR, Plan etc.
Operations Center	A center where one or more users may log in to operate the Chart II system. Operations centers are assigned responsibility for shared resources that are controlled by users who are logged in at that operations center.
Operator	A Chart II user that works at an Operations Center.
Organization	An organization is an agency that participates in the CHART II system and owns one or more Shared Resources.
Plan	A collection of plan items that can be added to the response plan of a traffic event as a group.
Plan Item	An action in the system that can be set up in advance to be activated one or more times in the future. Plan items must be contained in a plan. Specific types of plan items exist for specific functionality. A plan item may be copied to a traffic event response plan and subsequently activated.
Response Plan	A collection of response plan items created in response to a traffic event that can be activated as a group..
Response Plan Item	An action in the system that can be set up in response to a traffic event. Response plan items must be contained in a response plan. Specific types of response plan items exist for specific functionality. A response plan item carries out its specific task when activated
Role	A Role is a collection of functional rights that a user may perform. The roles that pertain to a particular user for a particular login session are determined when he/she logs into the system.
Safety Message Event	A Traffic Event that is entered by an Operator to display and/or broadcast safety messages.
Service Application	A software application that can be configured to run one or more service application modules and provides them basic services needed to serve CORBA objects.
Service Application Module	A software module that serves a related group of CORBA objects and can be run within the context of a service application.

Shared Resource	A resource that is owned by an organization. A user may be granted access to a shared resource owned by an organization through the functional rights scheme.
SHAZAM	A device used to notify the traveling public of the broadcast of a HAR message.
Sign	see DMS
Stored Message	A message that may be broadcast on a HAR or displayed on a DMS.
System Profile	Information used to define the configuration of the system. Properties stored in the system profile apply to all users when they are logged in.
Token	A token or access token is a security blob that encloses information about a user and the functional rights associated with the user. All secured Chart2 operations require a token to be passed to it and based on the functional rights found in a token a user is allowed or denied access.
Traffic Event	A traffic event represents a roadway event that is affecting traffic conditions and requires action from system operators.
Transferable Shared Resource	A shared resource that can be transferred from one operations center to another by a user with the appropriate functional rights.
User	A user is somebody who uses the CHART II system. A user can perform different operations in the system depending upon the roles they have been granted.
User Profile	A set of information used to correctly configure an individual user’s GUI on startup.
Weather Service Alert Event	A Traffic Event that is entered by an Operator in response to National Weather Service advisories.